NP-completeness

(or how to prove that problems are probably hard)

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

19 November 2013

Motivation

- This course is mostly about efficient algorithms and data structures for solving computational problems.
- Today we take a break from this and look at whether we can prove that a problem has no efficient algorithm.

Motivation

- This course is mostly about efficient algorithms and data structures for solving computational problems.
- Today we take a break from this and look at whether we can prove that a problem has no efficient algorithm.
- Why? Proving that a task is impossible can be helpful information, as it stops us from trying to complete it.

Motivation

- This course is mostly about efficient algorithms and data structures for solving computational problems.
- Today we take a break from this and look at whether we can prove that a problem has no efficient algorithm.
- Why? Proving that a task is impossible can be helpful information, as it stops us from trying to complete it.
- During this lecture we'll take an informal approach to discussing this, and computational complexity in general – see the references at the end for more detail.

Efficiency

Our first task is to define what we mean by efficiency.

Efficiency

Our first task is to define what we mean by efficiency.

- We think of an algorithm as being efficient if it runs in time polynomial in the input size.
- That is, if the input is n bits in size, the algorithm should run in time $O(n^c)$ for some constant c which does not depend on the input.

Efficiency

Our first task is to define what we mean by efficiency.

- We think of an algorithm as being efficient if it runs in time polynomial in the input size.
- That is, if the input is n bits in size, the algorithm should run in time $O(n^c)$ for some constant c which does not depend on the input.
- Examples of polynomial-time algorithms include Dijkstra's algorithm, Kruskal's algorithm, and in fact every algorithm you have seen in this course so far.

Efficiency

Our first task is to define what we mean by efficiency.

- We think of an algorithm as being efficient if it runs in time polynomial in the input size.
- That is, if the input is n bits in size, the algorithm should run in time $O(n^c)$ for some constant c which does not depend on the input.
- Examples of polynomial-time algorithms include Dijkstra's algorithm, Kruskal's algorithm, and in fact every algorithm you have seen in this course so far.
- An example of an algorithm which is not polynomial-time: testing whether an integer N is prime by trying to divide it by all integers m between 2 and $\sqrt{N}$.

Efficiency

Our first task is to define what we mean by efficiency.

- We think of an algorithm as being efficient if it runs in time polynomial in the input size.
- That is, if the input is n bits in size, the algorithm should run in time $O(n^c)$ for some constant c which does not depend on the input.
- Examples of polynomial-time algorithms include Dijkstra's algorithm, Kruskal's algorithm, and in fact every algorithm you have seen in this course so far.
- An example of an algorithm which is not polynomial-time: testing whether an integer N is prime by trying to divide it by all integers m between 2 and $\sqrt{N}$.
- As N is specified by $O(\log N)$ bits, this algorithm runs in time exponential in the input size.

Complexity classes

- For the rest of this lecture we will restrict to decision problems, i.e. problems with a yes/no answer.
- When we say “problem”, we really mean a family of problems, rather than just one instance.

Complexity classes

- For the rest of this lecture we will restrict to decision problems, i.e. problems with a yes/no answer.
- When we say “problem”, we really mean a family of problems, rather than just one instance.

Examples of decision problems:

- CONNECTIVITY: decide whether a graph is connected;
- PRIMALITY: decide whether an integer is prime;
- EDIT DISTANCE: given two strings and an integer k, decide whether their edit distance is at most k.

Complexity classes

- For the rest of this lecture we will restrict to decision problems, i.e. problems with a yes/no answer.
- When we say "problem", we really mean a family of problems, rather than just one instance.

Examples of decision problems:

- CONNECTIVITY: decide whether a graph is connected;
- PRIMALITY: decide whether an integer is prime;
- EDIT DISTANCE: given two strings and an integer k, decide whether their edit distance is at most k.

The set of decision problems which have algorithms with runtime polynomial in the input size is known as P.

- So we think of P as the class of decision problems which can be solved efficiently.

Formalities

Some notes about formalising this notion (which we'll largely ignore for the rest of this lecture):

- A decision problem can be formally identified with a language, i.e. a subset $\mathcal{L} \subseteq \{0,1\}^*$, where $\{0,1\}^*$ is the set of bit-strings of arbitrary length.
- Each input bit-string x such that $x \in \mathcal{L}$ corresponds to an input such that the answer should be "yes"; all strings $x \notin \mathcal{L}$ correspond to inputs such that the answer should be "no".

Formalities

Some notes about formalising this notion (which we'll largely ignore for the rest of this lecture):

- A decision problem can be formally identified with a language, i.e. a subset $\mathcal{L} \subseteq \{0,1\}^*$, where $\{0,1\}^*$ is the set of bit-strings of arbitrary length.
- Each input bit-string x such that $x \in \mathcal{L}$ corresponds to an input such that the answer should be "yes"; all strings $x \notin \mathcal{L}$ correspond to inputs such that the answer should be "no".
- The notion of "algorithm" should also be defined formally, in terms of Turing machines. However, we omit the details for this lecture.

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

- Imagine we want to solve a problem $\mathcal{L}_1$, but only know how to solve another problem $\mathcal{L}_2$. One way to solve $\mathcal{L}_1$ is simply to transform it into $\mathcal{L}_2$.

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

- Imagine we want to solve a problem $\mathcal{L}_1$, but only know how to solve another problem $\mathcal{L}_2$. One way to solve $\mathcal{L}_1$ is simply to transform it into $\mathcal{L}_2$.
- That is, imagine we have a polynomial-time algorithm which, given an instance of $\mathcal{L}_1$, transforms it into an instance of $\mathcal{L}_2$ such that the answer on the second instance is "yes" if and only if the answer on the first instance is "yes".

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

- Imagine we want to solve a problem $\mathcal{L}_1$, but only know how to solve another problem $\mathcal{L}_2$. One way to solve $\mathcal{L}_1$ is simply to transform it into $\mathcal{L}_2$.
- That is, imagine we have a polynomial-time algorithm which, given an instance of $\mathcal{L}_1$, transforms it into an instance of $\mathcal{L}_2$ such that the answer on the second instance is "yes" if and only if the answer on the first instance is "yes".
- Then we can use our algorithm for $\mathcal{L}_2$ to solve the second instance.

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

- Imagine we want to solve a problem $\mathcal{L}_1$, but only know how to solve another problem $\mathcal{L}_2$. One way to solve $\mathcal{L}_1$ is simply to transform it into $\mathcal{L}_2$.
- That is, imagine we have a polynomial-time algorithm which, given an instance of $\mathcal{L}_1$, transforms it into an instance of $\mathcal{L}_2$ such that the answer on the second instance is "yes" if and only if the answer on the first instance is "yes".
- Then we can use our algorithm for $\mathcal{L}_2$ to solve the second instance.
- We say that $\mathcal{L}_1$ reduces to $\mathcal{L}_2$ if such a transformation exists.

Reductions

One of the most fundamental questions in computer science is to determine which problems are in P. This can be done using the notion of reductions.

- Imagine we want to solve a problem $\mathcal{L}_1$, but only know how to solve another problem $\mathcal{L}_2$. One way to solve $\mathcal{L}_1$ is simply to transform it into $\mathcal{L}_2$.
- That is, imagine we have a polynomial-time algorithm which, given an instance of $\mathcal{L}_1$, transforms it into an instance of $\mathcal{L}_2$ such that the answer on the second instance is "yes" if and only if the answer on the first instance is "yes".
- Then we can use our algorithm for $\mathcal{L}_2$ to solve the second instance.
- We say that $\mathcal{L}_1$ reduces to $\mathcal{L}_2$ if such a transformation exists.
- If $\mathcal{L}_2 \in \mathrm{P}$, and $\mathcal{L}_1$ reduces to $\mathcal{L}_2$, then $\mathcal{L}_1 \in \mathrm{P}$.

Verifying solutions to problems

- There are many problems which we may not know how to solve efficiently, but for which, if someone claims the answer is yes, we can verify the claim efficiently.

Verifying solutions to problems

- There are many problems which we may not know how to solve efficiently, but for which, if someone claims the answer is yes, we can verify the claim efficiently.

- For example, consider the FACTORISATION problem: given two integers n and k, does n have a prime factor less than k?

Verifying solutions to problems

- There are many problems which we may not know how to solve efficiently, but for which, if someone claims the answer is yes, we can verify the claim efficiently.

- For example, consider the FACTORISATION problem: given two integers n and k, does n have a prime factor less than k?

- An instance of FACTORISATION: $n = 820\,580\,620\,832\,258\,609$, $k = 364\,797\,008$. Is the answer “yes”?

Verifying solutions to problems

- There are many problems which we may not know how to solve efficiently, but for which, if someone claims the answer is yes, we can verify the claim efficiently.
- For example, consider the FACTORISATION problem: given two integers n and k, does n have a prime factor less than k?
- An instance of FACTORISATION: $n = 820\,580\,620\,832\,258\,609$, $k = 364\,797\,008$. Is the answer "yes"?
- If someone claims the answer is "yes", they can give us the prime factors of n. We can then easily check whether they multiply together to give n, and whether one of them is less than k.

Verifying solutions to problems

- There are many problems which we may not know how to solve efficiently, but for which, if someone claims the answer is yes, we can verify the claim efficiently.
- For example, consider the FACTORISATION problem: given two integers n and k, does n have a prime factor less than k?
- An instance of FACTORISATION: $n = 820\,580\,620\,832\,258\,609$, $k = 364\,797\,008$. Is the answer "yes"?
- If someone claims the answer is "yes", they can give us the prime factors of n. We can then easily check whether they multiply together to give n, and whether one of them is less than k.
- Here the answer is indeed yes.

Verifying solutions to problems

The class of decision problems for which, if the answer is "yes", we can verify this in time polynomial in the input size is known as NP.

Verifying solutions to problems

The class of decision problems for which, if the answer is “yes”, we can verify this in time polynomial in the input size is known as NP.

- We can think of problems in this class as a game where we are given a proof that the answer is “yes” by an all-powerful wizard, and our job is to check that the proof is correct.

Verifying solutions to problems

The class of decision problems for which, if the answer is “yes”, we can verify this in time polynomial in the input size is known as NP.

- We can think of problems in this class as a game where we are given a proof that the answer is “yes” by an all-powerful wizard, and our job is to check that the proof is correct.
- If the answer is “yes”, we should accept a correct proof; but if the answer is “no”, we should not be fooled by any incorrect proof.

Verifying solutions to problems

The class of decision problems for which, if the answer is “yes”, we can verify this in time polynomial in the input size is known as NP.

- We can think of problems in this class as a game where we are given a proof that the answer is “yes” by an all-powerful wizard, and our job is to check that the proof is correct.
- If the answer is “yes”, we should accept a correct proof; but if the answer is “no”, we should not be fooled by any incorrect proof.
- It is clear that $\mathrm{P} \subseteq \mathrm{NP}$, as if we can solve a problem in polynomial time, we can efficiently verify a claimed solution we are given: we just ignore it, and solve the problem ourselves.

Verifying solutions to problems

The class of decision problems for which, if the answer is "yes", we can verify this in time polynomial in the input size is known as NP.

- We can think of problems in this class as a game where we are given a proof that the answer is "yes" by an all-powerful wizard, and our job is to check that the proof is correct.
- If the answer is "yes", we should accept a correct proof; but if the answer is "no", we should not be fooled by any incorrect proof.
- It is clear that $\mathrm{P} \subseteq \mathrm{NP}$, as if we can solve a problem in polynomial time, we can efficiently verify a claimed solution we are given: we just ignore it, and solve the problem ourselves.
- But whether or not $\mathrm{P} = \mathrm{NP}$ (aka the P vs. NP question) is the biggest unsolved problem in computer science!

More on NP

- The initials “NP” stand for Nondeterministic Polynomial (for reasons beyond the scope of this lecture. . .), and not Non-Polynomial.
- Indeed, the P vs. NP question precisely asks whether all problems in NP have polynomial-time algorithms.

More on NP

- The initials “NP” stand for Nondeterministic Polynomial (for reasons beyond the scope of this lecture. . .), and not Non-Polynomial.
- Indeed, the P vs. NP question precisely asks whether all problems in NP have polynomial-time algorithms.
- Resolving P vs. NP would win you everlasting fame (as well as $1M from the Clay Mathematics Institute).

More on NP

- The initials “NP” stand for Nondeterministic Polynomial (for reasons beyond the scope of this lecture...), and not Non-Polynomial.
- Indeed, the P vs. NP question precisely asks whether all problems in NP have polynomial-time algorithms.
- Resolving P vs. NP would win you everlasting fame (as well as $1M from the Clay Mathematics Institute).
- Although we don't know whether P $=$ NP, most people consider this very unlikely, as it would imply that whenever we have an efficient algorithm to verify a “yes” solution to a decision problem, we also have an efficient algorithm to solve the problem.

NP-hardness and NP-completeness

- We say that a decision problem $\mathcal{L}$ is NP-hard if, for every problem $\mathcal{L}' \in$ NP, there is a polynomial-time reduction from $\mathcal{L}'$ to $\mathcal{L}$.
- So, informally, NP-hard problems are at least as hard as the hardest problems in NP.

NP-hardness and NP-completeness

- We say that a decision problem $\mathcal{L}$ is NP-hard if, for every problem $\mathcal{L}' \in$ NP, there is a polynomial-time reduction from $\mathcal{L}'$ to $\mathcal{L}$.
- So, informally, NP-hard problems are at least as hard as the hardest problems in NP.
- If $\mathcal{L}$ is NP-hard, and there exists a polynomial-time algorithm for $\mathcal{L}$, then P $=$ NP. So if we can prove that $\mathcal{L}$ is NP-hard, this is evidence that there is no polynomial-time algorithm that solves it.

NP-hardness and NP-completeness

- We say that a decision problem $\mathcal{L}$ is NP-hard if, for every problem $\mathcal{L}' \in$ NP, there is a polynomial-time reduction from $\mathcal{L}'$ to $\mathcal{L}$.

- So, informally, NP-hard problems are at least as hard as the hardest problems in NP.

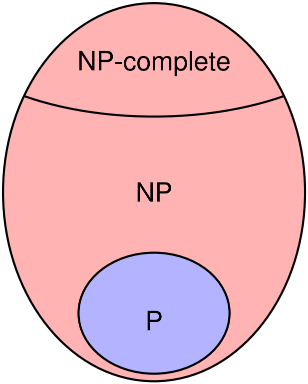
- If $\mathcal{L}$ is NP-hard, and there exists a polynomial-time algorithm for $\mathcal{L}$, then P $=$ NP. So if we can prove that $\mathcal{L}$ is NP-hard, this is evidence that there is no polynomial-time algorithm that solves it.

- We say that a problem $\mathcal{L}$ is NP-complete if $\mathcal{L}$ is NP-hard and $\mathcal{L} \in$ NP. Informally, NP-complete problems are the hardest problems in NP.

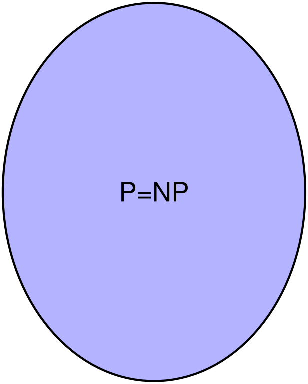
- It is not obvious that any NP-complete problems should exist...

P and NP in pictures

The picture if P≠NP:



The picture if P=NP:



An NP-complete problem

The CIRCUIT SAT (short for "satisfiability") problem is defined as follows.

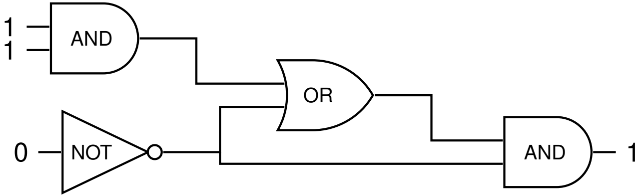
- The input to the problem is a circuit (i.e. a sequence of AND, OR and NOT gates connected by wires in some order).
- The circuit takes some bits as input and produces a single-bit output.
- The problem is to determine whether there exists an input such that the output is 1.

An NP-complete problem

The CIRCUIT SAT (short for "satisfiability") problem is defined as follows.

- The input to the problem is a circuit (i.e. a sequence of AND, OR and NOT gates connected by wires in some order).
- The circuit takes some bits as input and produces a single-bit output.
- The problem is to determine whether there exists an input such that the output is 1.

For example:

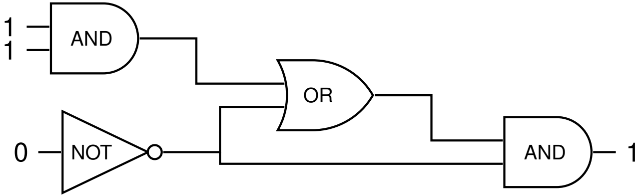


An NP-complete problem

The CIRCUIT SAT (short for "satisfiability") problem is defined as follows.

- The input to the problem is a circuit (i.e. a sequence of AND, OR and NOT gates connected by wires in some order).
- The circuit takes some bits as input and produces a single-bit output.
- The problem is to determine whether there exists an input such that the output is 1.

For example:



CIRCUIT SAT is in NP: if the answer is "yes", and we are given a claimed input such that the output is 1, we can simulate the circuit to check it.

An NP-complete problem

Claim

CIRCUIT SAT is NP-hard.

An NP-complete problem

Claim

CIRCUIT SAT is NP-hard.

Proof sketch

- We need to show that, for any $\mathcal{L} \in$ NP, $\mathcal{L}$ reduces to CIRCUIT SAT.

An NP-complete problem

Claim

CIRCUIT SAT is NP-hard.

Proof sketch

- We need to show that, for any $\mathcal{L} \in$ NP, $\mathcal{L}$ reduces to CIRCUIT SAT.
- If $\mathcal{L} \in$ NP, then there is a polynomial-time algorithm which checks a claimed solution to $\mathcal{L}$ when the answer is "yes".

An NP-complete problem

Claim

CIRCUIT SAT is NP-hard.

Proof sketch

- We need to show that, for any $\mathcal{L} \in$ NP, $\mathcal{L}$ reduces to CIRCUIT SAT.
- If $\mathcal{L} \in$ NP, then there is a polynomial-time algorithm which checks a claimed solution to $\mathcal{L}$ when the answer is "yes".
- We can write any such algorithm as a circuit with at most polynomially many gates by "compiling" it.

An NP-complete problem

Claim

Circuit SAT is NP-hard.

Proof sketch

- We need to show that, for any $\mathcal{L} \in$ NP, $\mathcal{L}$ reduces to Circuit SAT.
- If $\mathcal{L} \in$ NP, then there is a polynomial-time algorithm which checks a claimed solution to $\mathcal{L}$ when the answer is "yes".
- We can write any such algorithm as a circuit with at most polynomially many gates by "compiling" it.
- If there exists a proof that the answer should be "yes", this corresponds to an input to the circuit such that the output is 1; otherwise, there is no such input.

An NP-complete problem

Claim

CIRCUIT SAT is NP-hard.

Proof sketch

- We need to show that, for any $\mathcal{L} \in$ NP, $\mathcal{L}$ reduces to CIRCUIT SAT.
- If $\mathcal{L} \in$ NP, then there is a polynomial-time algorithm which checks a claimed solution to $\mathcal{L}$ when the answer is "yes".
- We can write any such algorithm as a circuit with at most polynomially many gates by "compiling" it.
- If there exists a proof that the answer should be "yes", this corresponds to an input to the circuit such that the output is 1; otherwise, there is no such input.
- So, if we can solve CIRCUIT SAT, we can decide which of these is the case. □

More NP-complete problems

- Now that we know that Circuit SAT is NP-complete, we can use this to prove that other problems are also NP-complete.
- If we have a problem $\mathcal{L} \in \mathrm{NP}$ such that Circuit SAT reduces to $\mathcal{L}$, then $\mathcal{L}$ must be NP-complete.

More NP-complete problems

- Now that we know that CIRCUIT SAT is NP-complete, we can use this to prove that other problems are also NP-complete.
- If we have a problem $\mathcal{L} \in$ NP such that CIRCUIT SAT reduces to $\mathcal{L}$, then $\mathcal{L}$ must be NP-complete.
- This can be seen as good evidence that there is no efficient algorithm for $\mathcal{L}$.

More NP-complete problems

- Now that we know that CIRCUIT SAT is NP-complete, we can use this to prove that other problems are also NP-complete.
- If we have a problem $\mathcal{L} \in$ NP such that CIRCUIT SAT reduces to $\mathcal{L}$, then $\mathcal{L}$ must be NP-complete.
- This can be seen as good evidence that there is no efficient algorithm for $\mathcal{L}$.
- The first problem for which we will prove NP-completeness in this way is called 3-SAT. This is the problem of determining, given a boolean formula in conjunctive normal form with at most 3 variables per clause, whether it has a satisfying assignment.

More NP-complete problems

- Now that we know that CIRCUIT SAT is NP-complete, we can use this to prove that other problems are also NP-complete.
- If we have a problem $\mathcal{L} \in \text{NP}$ such that CIRCUIT SAT reduces to $\mathcal{L}$, then $\mathcal{L}$ must be NP-complete.
- This can be seen as good evidence that there is no efficient algorithm for $\mathcal{L}$.
- The first problem for which we will prove NP-completeness in this way is called 3-SAT. This is the problem of determining, given a boolean formula in conjunctive normal form with at most 3 variables per clause, whether it has a satisfying assignment.
- What does this mean?

3-SAT

- A boolean formula in conjunctive normal form is an expression of the form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

where each c_i is a clause and the $\wedge$'s mean AND.

3-SAT

- A boolean formula in conjunctive normal form is an expression of the form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

  where each c_i is a clause and the $\wedge$'s mean AND.
- A clause is the OR ("$\vee$") of variables $x_i \in \{0, 1\}$ or their negations $\neg x_i$ (where $\neg$ means NOT), for example:

$$(x_3 \vee \neg x_2 \vee x_7)$$

3-SAT

- A boolean formula in conjunctive normal form is an expression of the form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

  where each c_i is a clause and the $\wedge$'s mean AND.

- A clause is the OR ("$\vee$") of variables $x_i \in \{0, 1\}$ or their negations $\neg x_i$ (where $\neg$ means NOT), for example:

$$(x_3 \vee \neg x_2 \vee x_7)$$

- A satisfying assignment is an assignment to the variables such that the whole formula evaluates to 1 (true).

3-SAT

- A boolean formula in conjunctive normal form is an expression of the form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

  where each c_i is a clause and the $\wedge$'s mean AND.

- A clause is the OR ("$\vee$") of variables $x_i \in \{0, 1\}$ or their negations $\neg x_i$ (where $\neg$ means NOT), for example:

$$(x_3 \vee \neg x_2 \vee x_7)$$

- A satisfying assignment is an assignment to the variables such that the whole formula evaluates to 1 (true).

For example:

$$(x_2 \vee x_1 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_2 \vee x_3 \vee x_4)$$

is an instance of 3-SAT. It is satisfied by e.g. $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1$.

3-SAT

- In the 3-SAT problem we are given a boolean formula and asked to determine whether it has a satisfying assignment.
- The problem is in NP because, if someone claims there is a satisfying assignment to the formula, they can give us the assignment and we can check it efficiently.

3-SAT

- In the 3-SAT problem we are given a boolean formula and asked to determine whether it has a satisfying assignment.
- The problem is in NP because, if someone claims there is a satisfying assignment to the formula, they can give us the assignment and we can check it efficiently.
- But it's not so clear how to find a satisfying assignment efficiently ourselves; we could try each possible assignment one after the other, but there are 2^n possible assignments to n variables, so this could be very slow.

3-SAT

- In the 3-SAT problem we are given a boolean formula and asked to determine whether it has a satisfying assignment.
- The problem is in NP because, if someone claims there is a satisfying assignment to the formula, they can give us the assignment and we can check it efficiently.
- But it's not so clear how to find a satisfying assignment efficiently ourselves; we could try each possible assignment one after the other, but there are 2^n possible assignments to n variables, so this could be very slow.
- In fact, the best known algorithms for solving 3-SAT with n variables run in time $2^{\Omega(n)}$, i.e. take exponential time in n.

3-SAT

- In the 3-SAT problem we are given a boolean formula and asked to determine whether it has a satisfying assignment.
- The problem is in NP because, if someone claims there is a satisfying assignment to the formula, they can give us the assignment and we can check it efficiently.
- But it's not so clear how to find a satisfying assignment efficiently ourselves; we could try each possible assignment one after the other, but there are 2^n possible assignments to n variables, so this could be very slow.
- In fact, the best known algorithms for solving 3-SAT with n variables run in time $2^{\Omega(n)}$, i.e. take exponential time in n.
- It turns out that 3-SAT is actually NP-complete.

Proof that 3-SAT is NP-complete (sketch)

- We will reduce CIRCUIT SAT to 3-SAT.

Proof that 3-SAT is NP-complete (sketch)

- We will reduce CIRCUIT SAT to 3-SAT.
- We need to show that, given a circuit, we can transform it into a boolean formula such that the formula is satisfiable if and only if there is an input to the circuit such that it outputs 1.

Proof that 3-SAT is NP-complete (sketch)

- We will reduce CIRCUIT SAT to 3-SAT.
- We need to show that, given a circuit, we can transform it into a boolean formula such that the formula is satisfiable if and only if there is an input to the circuit such that it outputs 1.
- We use a construction where each wire in the circuit corresponds to a variable in the formula, and there are several clauses for each gate.
- For each gate, there exists an assignment to the variables satisfying the clauses if and only if the gate behaves correctly.

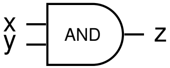
Proof that 3-SAT is NP-complete (sketch)

- We will reduce CIRCUIT SAT to 3-SAT.
- We need to show that, given a circuit, we can transform it into a boolean formula such that the formula is satisfiable if and only if there is an input to the circuit such that it outputs 1.
- We use a construction where each wire in the circuit corresponds to a variable in the formula, and there are several clauses for each gate.
- For each gate, there exists an assignment to the variables satisfying the clauses if and only if the gate behaves correctly.
- Finally, we have a clause containing a single variable, which is satisfied if and only if the output wire of the circuit is set to 1.

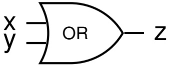
...

Proof that 3-SAT is NP-complete (sketch)

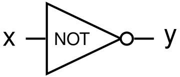
- The construction performs the mapping:



$$\mapsto \quad (z \vee \neg x \vee \neg y) \wedge (\neg z \vee x) \wedge (\neg z \vee y)$$



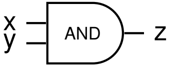
$$\mapsto \quad (\neg z \vee x \vee y) \wedge (z \vee \neg x) \wedge (z \vee \neg y)$$



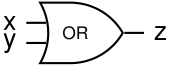
$$\mapsto \quad (x \vee y) \wedge (\neg x \vee \neg y)$$

Proof that 3-SAT is NP-complete (sketch)

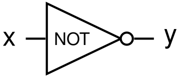
- The construction performs the mapping:



$$\mapsto \quad (z \vee \neg x \vee \neg y) \wedge (\neg z \vee x) \wedge (\neg z \vee y)$$



$$\mapsto \quad (\neg z \vee x \vee y) \wedge (z \vee \neg x) \wedge (z \vee \neg y)$$



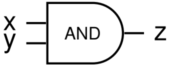
$$\mapsto \quad (x \vee y) \wedge (\neg x \vee \neg y)$$

- For example, $y = \neg x$ if and only if $(x \vee y) = 1$ and $(\neg x \vee \neg y) = 1$.

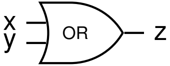
Ashley Montanaro
ashley@cs.bris.ac.uk

Proof that 3-SAT is NP-complete (sketch)

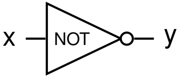
- The construction performs the mapping:



$$\mapsto \quad (z \vee \neg x \vee \neg y) \wedge (\neg z \vee x) \wedge (\neg z \vee y)$$



$$\mapsto \quad (\neg z \vee x \vee y) \wedge (z \vee \neg x) \wedge (z \vee \neg y)$$



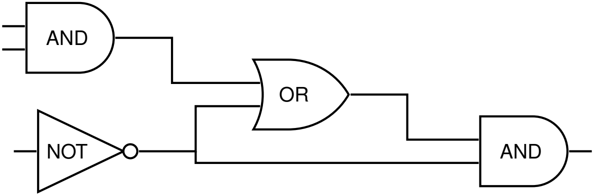
$$\mapsto \quad (x \vee y) \wedge (\neg x \vee \neg y)$$

- For example, $y = \neg x$ if and only if $(x \vee y) = 1$ and $(\neg x \vee \neg y) = 1$.

- Claim: All the clauses are satisfied if and only if all the gates work properly, and the output of the circuit is 1.

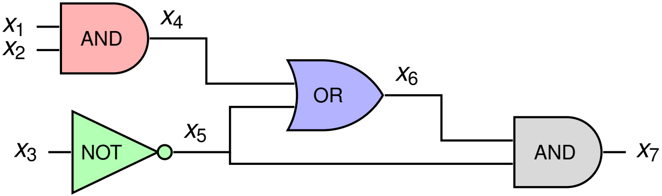
Example

Imagine we want to solve CIRCUIT SAT for the following circuit:



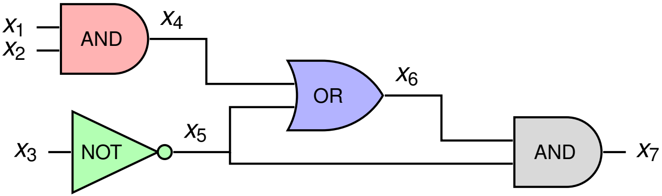
Example

Imagine we want to solve CIRCUIT SAT for the following circuit:



Example

Imagine we want to solve CIRCUIT SAT for the following circuit:

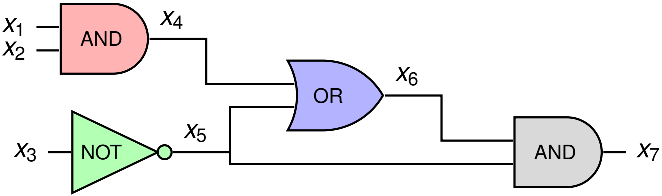


This maps to the following formula:

$$
\begin{aligned}
& (x_4 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_4 \vee x_1) \wedge (\neg x_4 \vee x_2) \\
\wedge \quad & (x_3 \vee x_5) \wedge (\neg x_3 \vee \neg x_5) \\
\wedge \quad & (\neg x_6 \vee x_4 \vee x_5) \wedge (x_6 \vee \neg x_4) \wedge (x_6 \vee \neg x_5) \\
\wedge \quad & (x_7 \vee \neg x_6 \vee \neg x_5) \wedge (\neg x_7 \vee x_6) \wedge (\neg x_7 \vee x_5) \wedge (x_7)
\end{aligned}
$$

Example

Imagine we want to solve CIRCUIT SAT for the following circuit:



This maps to the following formula:

$$
\begin{aligned}
& (x_4 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_4 \vee x_1) \wedge (\neg x_4 \vee x_2) \\
\wedge \quad & (x_3 \vee x_5) \wedge (\neg x_3 \vee \neg x_5) \\
\wedge \quad & (\neg x_6 \vee x_4 \vee x_5) \wedge (x_6 \vee \neg x_4) \wedge (x_6 \vee \neg x_5) \\
\wedge \quad & (x_7 \vee \neg x_6 \vee \neg x_5) \wedge (\neg x_7 \vee x_6) \wedge (\neg x_7 \vee x_5) \wedge (x_7)
\end{aligned}
$$

The formula is satisfiable, so the original circuit is too.

Another NP-complete problem: 3-COLOURING

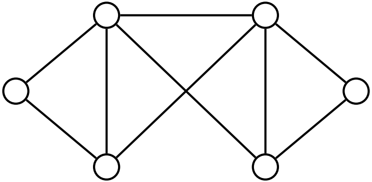
- We will now show NP-completeness of another problem, which is apparently quite different: graph colouring.

- The 3-COLOURING problem is defined as follows: Given an undirected graph G, determine whether each vertex of G can be coloured with one of three colours, such that any two vertices connected by an edge are assigned different colours.

Another NP-complete problem: 3-COLOURING

- We will now show NP-completeness of another problem, which is apparently quite different: graph colouring.
- The 3-COLOURING problem is defined as follows: Given an undirected graph G, determine whether each vertex of G can be coloured with one of three colours, such that any two vertices connected by an edge are assigned different colours.

For example:

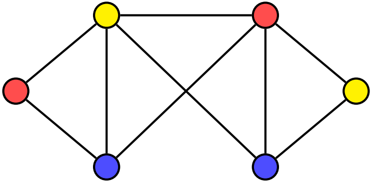


Ashley Montanaro
ashley@cs.bris.ac.uk

Another NP-complete problem: 3-COLOURING

- We will now show NP-completeness of another problem, which is apparently quite different: graph colouring.
- The 3-COLOURING problem is defined as follows: Given an undirected graph G, determine whether each vertex of G can be coloured with one of three colours, such that any two vertices connected by an edge are assigned different colours.

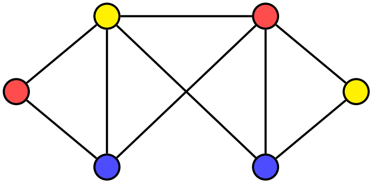
For example:



Another NP-complete problem: 3-COLOURING

- We will now show NP-completeness of another problem, which is apparently quite different: graph colouring.
- The 3-COLOURING problem is defined as follows: Given an undirected graph G, determine whether each vertex of G can be coloured with one of three colours, such that any two vertices connected by an edge are assigned different colours.

For example:



3-COLOURING is in NP because, if someone gives us a claimed colouring of a graph, we can check it efficiently.

Proof that 3-COLOURING is NP-complete (sketch)

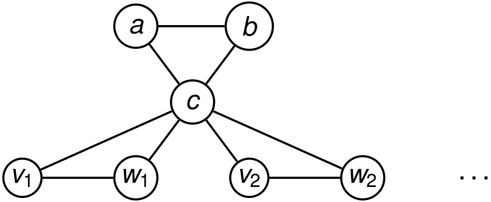
- We prove that 3-COLOURING is NP-complete by reducing 3-SAT to 3-COLOURING.

Proof that 3-COLOURING is NP-complete (sketch)

- We prove that 3-COLOURING is NP-complete by reducing 3-SAT to 3-COLOURING.
- Given a boolean formula, the idea is to create a graph with vertices corresponding to variables, and edges corresponding to clauses, such that the graph is colourable with 3 colours if and only if the formula is satisfiable.

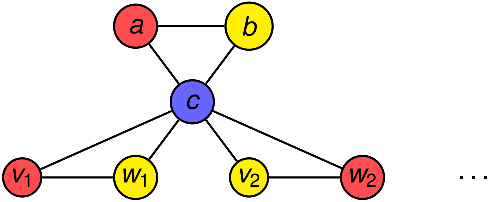
Proof that 3-COLOURING is NP-complete (sketch)

- We prove that 3-COLOURING is NP-complete by reducing 3-SAT to 3-COLOURING.
- Given a boolean formula, the idea is to create a graph with vertices corresponding to variables, and edges corresponding to clauses, such that the graph is colourable with 3 colours if and only if the formula is satisfiable.
- We start by having a pair of vertices v_i, w_i for each variable x_i in the formula. Each of these vertices is connected to a central vertex c, which is connected in turn to two other vertices a and b.



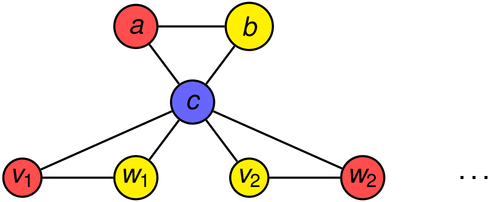
Proof that 3-COLOURING is NP-complete (sketch)

- Imagine (without loss of generality) that vertices a, b and c are coloured red, yellow and blue.
- Then all of the pairs of vertices v_i, w_i must be coloured red and yellow (one of them red, and the other yellow).



Proof that 3-COLOURING is NP-complete (sketch)

- Imagine (without loss of generality) that vertices a, b and c are coloured red, yellow and blue.
- Then all of the pairs of vertices v_i, w_i must be coloured red and yellow (one of them red, and the other yellow).



- This will be used to encode whether the i'th variable x_i is 0 or 1 in some assignment to the original formula.
- If v_i is red and w_i is yellow, this will correspond to $x_i = 0$; if v_i is yellow and w_i is red, this will correspond to $x_i = 1$.

Proof that 3-COLOURING is NP-complete (sketch)

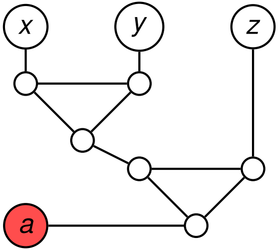
The second ingredient is a clause gadget.

- This is a subgraph which is only colourable correctly if at least one of three "incoming" vertices x, y, z is not coloured red.

Proof that 3-COLOURING is NP-complete (sketch)

The second ingredient is a clause gadget.

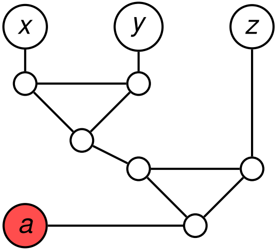
- This is a subgraph which is only colourable correctly if at least one of three "incoming" vertices x, y, z is not coloured red.
- The gadget looks like this:



Proof that 3-COLOURING is NP-complete (sketch)

The second ingredient is a clause gadget.

- This is a subgraph which is only colourable correctly if at least one of three "incoming" vertices x, y, z is not coloured red.
- The gadget looks like this:



- Claim: There is a valid 3-colouring of the internal (unlabelled) vertices if and only if at least one of x, y, z is not coloured red.

Proof that 3-COLOURING is NP-complete (sketch)

We now combine clause gadgets with the previous graph.

- For each clause, we connect the gadget to vertices corresponding to the variables that appear in that clause.

Proof that 3-COLOURING is NP-complete (sketch)

We now combine clause gadgets with the previous graph.

- For each clause, we connect the gadget to vertices corresponding to the variables that appear in that clause.
- For each variable x_i in a clause, if x_i is negated in the clause, we connect the gadget to vertex w_i; if it is not negated, we connect the gadget to v_i.
- This enforces the constraint that each clause must be satisfied – i.e. evaluate to 1.

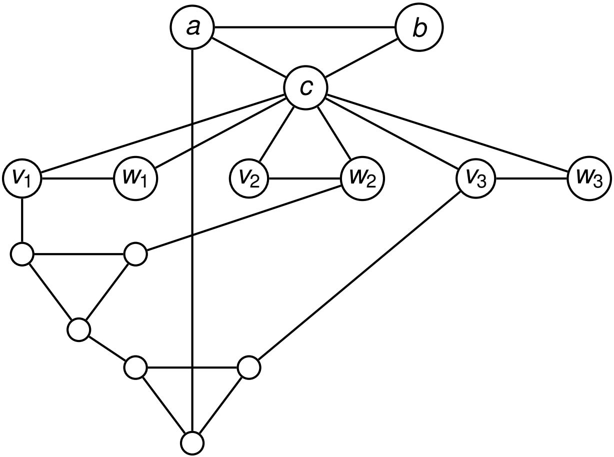
Proof that 3-COLOURING is NP-complete (sketch)

We now combine clause gadgets with the previous graph.

- For each clause, we connect the gadget to vertices corresponding to the variables that appear in that clause.
- For each variable x_i in a clause, if x_i is negated in the clause, we connect the gadget to vertex w_i; if it is not negated, we connect the gadget to v_i.
- This enforces the constraint that each clause must be satisfied – i.e. evaluate to 1.
- Claim: Any valid colouring of the graph corresponds to an assignment to the variables such that all clauses are satisfied.
- This means that determining whether the graph is 3-colourable allows us to determine whether the formula is satisfiable, so 3-COLOURING is NP-complete.

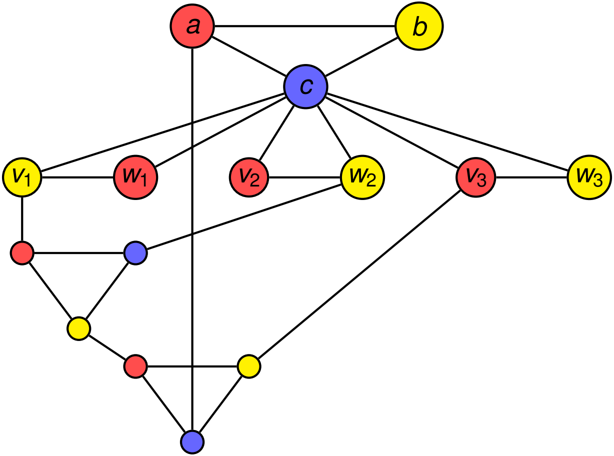
Example

The graph corresponding to the formula $(x_1 \vee \neg x_2 \vee x_3)$ is:



Example

The graph can be coloured properly, corresponding to the original formula having a satisfying assignment. One such colouring:



The colouring shown corresponds to assigning $x_1 = 1$, $x_2 = 0$, $x_3 = 0$.

Other NP-complete problems

A vast number of other problems have also been proven to be NP-complete, many of which are very important in science, engineering and business.

For example:

- Timetable scheduling
- Packing and covering problems
- Finding longest paths
- Solving systems of quadratic equations
- Partitioning problems
- Finding the longest common subsequence of two strings
- Many games and puzzles, e.g. generalised Sudoku and Lemmings
- Integer programming (see later in this course)
- …

Summary

- The theory of NP-completeness allows us to make rigorous the intuition that some problems are intrinsically hard.
- If a problem is NP-complete, this is good evidence that there is no efficient (polynomial-time) algorithm to solve it in the worst case.
- We can prove that a problem is NP-complete by showing that some other NP-complete problem reduces to it.

Summary

- The theory of NP-completeness allows us to make rigorous the intuition that some problems are intrinsically hard.
- If a problem is NP-complete, this is good evidence that there is no efficient (polynomial-time) algorithm to solve it in the worst case.
- We can prove that a problem is NP-complete by showing that some other NP-complete problem reduces to it.

What if we are faced with an NP-complete problem that we have to solve?

Summary

- The theory of NP-completeness allows us to make rigorous the intuition that some problems are intrinsically hard.
- If a problem is NP-complete, this is good evidence that there is no efficient (polynomial-time) algorithm to solve it in the worst case.
- We can prove that a problem is NP-complete by showing that some other NP-complete problem reduces to it.

What if we are faced with an NP-complete problem that we have to solve? There are several approaches we can take:

1. Find an efficient algorithm which works for the particular cases we care about;

Summary

- The theory of NP-completeness allows us to make rigorous the intuition that some problems are intrinsically hard.
- If a problem is NP-complete, this is good evidence that there is no efficient (polynomial-time) algorithm to solve it in the worst case.
- We can prove that a problem is NP-complete by showing that some other NP-complete problem reduces to it.

What if we are faced with an NP-complete problem that we have to solve? There are several approaches we can take:

1. Find an efficient algorithm which works for the particular cases we care about;
2. Find an efficient algorithm which outputs an approximate solution (see COMS31900: Advanced Algorithms for more);

Summary

- The theory of NP-completeness allows us to make rigorous the intuition that some problems are intrinsically hard.
- If a problem is NP-complete, this is good evidence that there is no efficient (polynomial-time) algorithm to solve it in the worst case.
- We can prove that a problem is NP-complete by showing that some other NP-complete problem reduces to it.

What if we are faced with an NP-complete problem that we have to solve? There are several approaches we can take:

1. Find an efficient algorithm which works for the particular cases we care about;
2. Find an efficient algorithm which outputs an approximate solution (see COMS31900: Advanced Algorithms for more);
3. Prove P=NP and win a million dollars.

Further Reading

- Introduction to Algorithms
  T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Chapter 34 – NP-completeness
- Algorithms
  S. Dasgupta, C. H. Papadimitriou and U. V. Vazirani
  `http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/`
  - Chapter 8 – NP-complete problems
- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`
  - Lecture 29 – NP-Hard Problems

Biographical notes

Stephen Cook (b. 1939)

- An American-Canadian mathematician who invented the notion of NP-completeness in a seminal paper in 1971.
- After this, many important problems were swiftly proven to be NP-complete.
- Cook won the Turing Award in 1982.
- Also has a computational complexity class named after him (SC).

Pic: Wikipedia

Biographical notes

Leonid Levin (b. 1948)

- Levin is a Soviet-American computer scientist who independently discovered the notion of NP-completeness.
- Neither Cook nor Levin were aware of the other's work due to the Iron Curtain.
- The fact that boolean satisfiability is NP-complete is now known as the Cook-Levin Theorem.

Pic: Wikipedia