Algorithmics

Lecture 8

Scan the QR code below or use the password listed below to take your attendance

6y5lyl

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 4 – NP-completeness

Introduction (examples and discussion)

NP-complete problems

The classes P and NP

Polynomial-time reductions

Formal definition of NP-completeness

How to prove a problem is NP-complete

Some efficient algorithms we have seen

We have seen algorithms for a wide range of problems so far, giving us a spectrum of worst-case complexity functions:

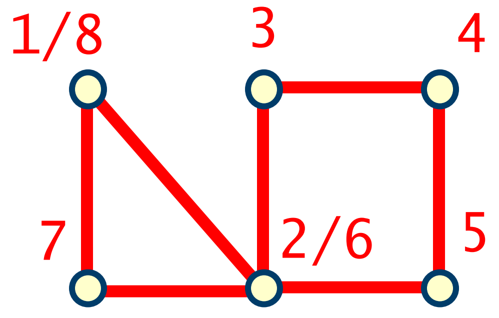
- searching a sorted list `O(log n)` (for an array/list of length **n**)
- finding the max value `O(n)` (for an array/list of length **n**)
- comparison-based sorting `O(n log n)` (for an array/list of length **n**)
- distance between two strings $O(n^2)$ (for two strings of length **n**)
- finding a shortest path $O(n^2)$ (for weighted graph with **n** vertices)

These are all examples of problems that admit polynomial-time algorithms: their worst-case complexity is $O(n^c)$ for some constant **c**

The Eulerian cycle problem

G undirected graph: decide whether G admits an Euler cycle

– an Eulerian cycle is a cycle that traverses each edge exactly once



Theorem (Euler, 1736). A connected undirected graph has an Euler cycle if and only if each vertex has even degree

therefore we can test whether G has an Euler cycle (and find one) in:

– $O(n^2)$ time if G is represented by an adjacency matrix

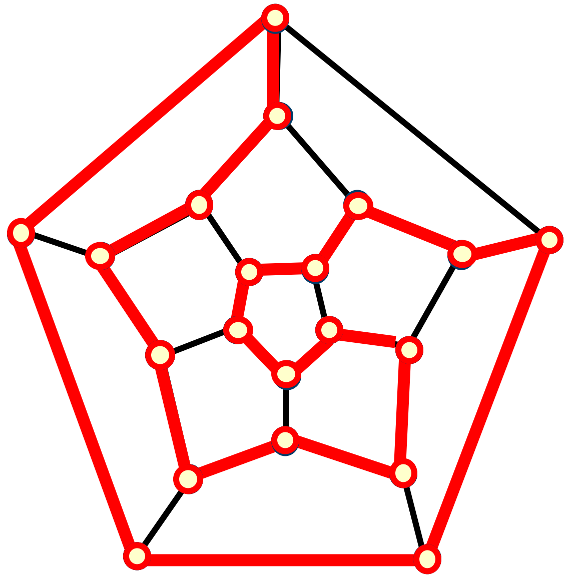
– $O(m+n)$ time if G is represented by adjacency lists

– where $m=|E|$ and $n=|V|$

The Hamiltonian cycle problem

G undirected graph, decide whether G admits an Hamiltonian cycle

- a Hamiltonian cycle is a cycle that visits each vertex exactly once



This problem is superficially similar to the Euler cycle problem

- however in an algorithmic sense it is very different
- nobody has found a polynomial-time algorithm for Hamiltonian cycle

The Hamiltonian cycle problem

Brute force algorithm:

- generate all permutations of vertices
- check each one to see if it is a cycle, i.e. corresponding edges are present

Complexity of the algorithm (n is the number of vertices)

- $n!$ permutations will be generated in the worst case
- for each permutation π, $O(n^2)$ operations to check whether π is a Hamiltonian cycle (assuming G is represented by adjacency lists)
  - worst case: to check an edge is present have to traverse adjacency list of length $n-1$ and have n edges to check

However: an example where using an adjacency matrix is more efficient as $O(n)$ operations are required to check a permutation

- can check each edge in $O(1)$
- just involves checking value in an array is 0 or 1

The Hamiltonian cycle problem

Brute force algorithm:

- generate all permutations of vertices
- check each one to see if it is a cycle, i.e. corresponding edges are present

Complexity of the algorithm (n is the number of vertices)

- $n!$ permutations will be generated in the worst case
- for each permutation π, $O(n^2)$ operations to check whether π is a Hamiltonian cycle (assuming G is represented by adjacency lists)
    - worst case: to check an edge is present have to traverse adjacency list of length $n-1$ and have n edges to check

Therefore worst-case number of operations is $O(n^2 n!)$

- this is an example of an exponential algorithm
- an algorithm whose time complexity is no better than $O(b^n)$ for some constant b (and so cannot be expressed as $O(n^c)$ for any constant c)

Polynomial versus exponential time

Table shows running time of algorithms with various complexities (assuming 10^9 operations per second)

	20	40	50	60	70
n	.00001 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.027 sec	.064 sec	.125 sec	.216 sec
n^5	.1 sec	24.3 secs	1.7 mins	5.2 mins	13.0 mins
2^n	.001 sec	17.9 mins	12.7 days	35.7 years	366 cents
3^n	.059 sec	6.5 years	3855 cents	2×10^8 cents	1.3×10^{13} cents
$n!$	3.6 secs	8.4×10^{16} cents	2.6×10^{32} cents	9.6×10^{48} cents	2.6×10^{66} cents

Polynomial versus exponential time

Table shows running time of algorithms with various complexities (assuming 10^9 operations per second)

	20	40	50	60	70
n	.00001 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.027 sec	.064 sec	.125 sec	.216 sec
n^5	.1 sec	24.3 secs	1.7 mins	5.2 mins	13.0 mins
2^n	.001 sec	17.9 mins	12.7 days	35.7 years	366 cents
3^n	.059 sec	6.5 years	3855 cents	2×10^8 cents	1.3×10^{13} cents
$n!$	3.6 secs	8.4×10^{16} cents	2.6×10^{32} cents	9.6×10^{48} cents	2.6×10^{66} cents

As **n** grows, distinction between polynomial and exponential time algorithms becomes dramatic

Polynomial versus exponential time

This behaviour still applies even with increases in computing power

- sizes of largest instance solvable in 1 hour on a current computer

	current computer
n	N_1
n^2	N_2
n^3	N_3
n^5	N_4
2^n	N_5
3^n	N_6
$n!$	N_7

Polynomial versus exponential time

This behaviour still applies even with increases in computing power

– sizes of largest instance solvable in 1 hour on a current computer
– what happens when computers become faster?

	current computer	computer 100 times faster	computer 1000 times faster
n	N_1		
n^2	N_2		
n^3	N_3		
n^5	N_4		
2^n	N_5		
3^n	N_6		
$n!$	N_7		

Polynomial versus exponential time

This behaviour still applies even with increases in computing power

– sizes of largest instance solvable in 1 hour on a current computer

– what happens when computers become faster?

	current computer	computer 100 times faster	computer 1000 times faster
n	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
n^3	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
n^5	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$
$n!$	N_7	$\leq N_7 + 1$	$\leq N_7 + 1$

Polynomial versus exponential time

This behaviour still applies even with increases in computing power

- sizes of largest instance solvable in 1 hour on a current computer
- what happens when computers become faster?

	current computer	computer 100 times faster	computer 1000 times faster
n	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
n^3	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
n^5	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	**N_6**	**$N_6 + 4.19$**	**$N_6 + 6.29$**
$n!$	N_7	$\leq N_7 + 1$	$\leq N_7 + 1$

A thousand-fold increase in computing power only adds 6 to the size of the largest problem instance solvable in 1 hour, for an algorithm with complexity 3^n

Polynomial versus exponential time

The message:

- **Exponential-time algorithms are in general "bad"**
  - increases in processor speeds do not lead to significant changes in this slow behaviour when the input size is large
- **Polynomial-time algorithms are in general "good"**

When we refer to "efficient algorithms" we mean polynomial-time

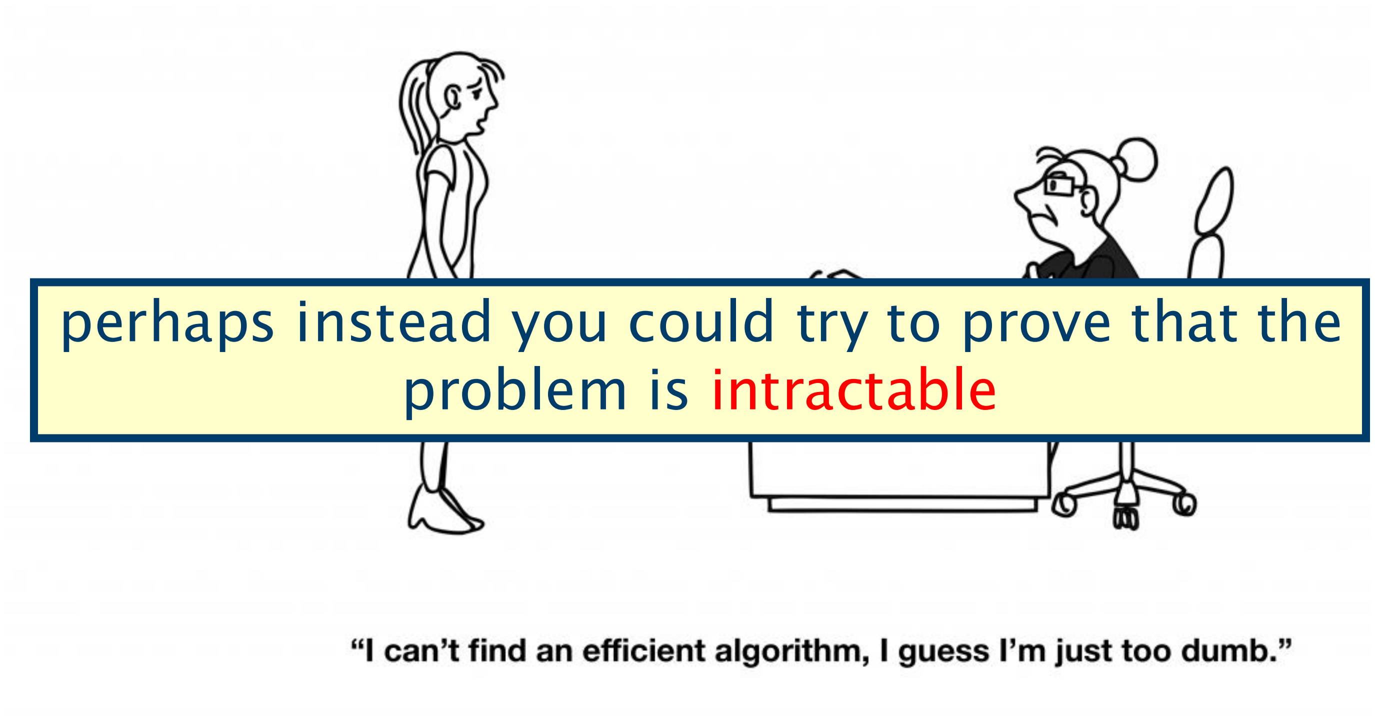
- often polynomial-time algorithms require some extra insight
- often exponential-time algorithms are variations on exhaustive search

A problem is polynomial-time solvable if it admits a polynomial-time algorithm

A brief interlude

You are asked to find a polynomial-time algorithm for the Hamiltonian cycle problem

– this could be a difficult task, you do not want to have to report:

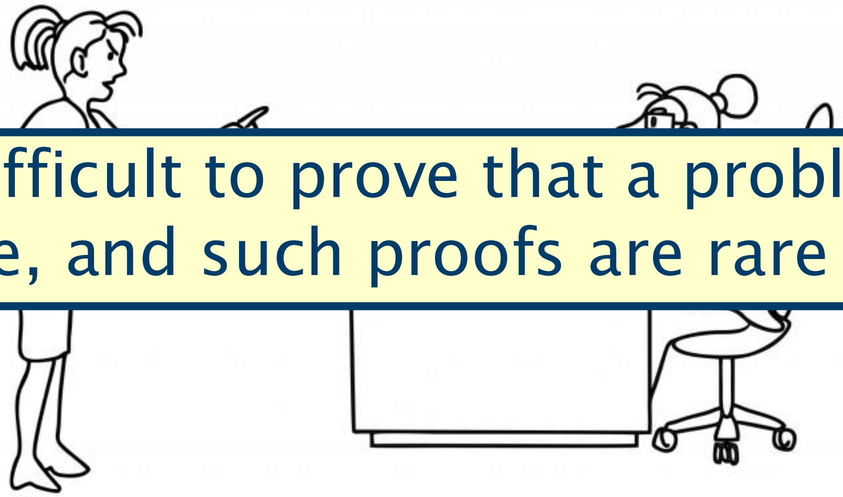

"I can't find an efficient algorithm, I guess I'm just too dumb."

perhaps instead you could try to prove that the problem is intractable

A brief interlude

Definition: a problem Π is intractable if there does not exist a polynomial-time algorithm that solves Π

- you could try to prove that the Hamiltonian Cycle problem is intractable

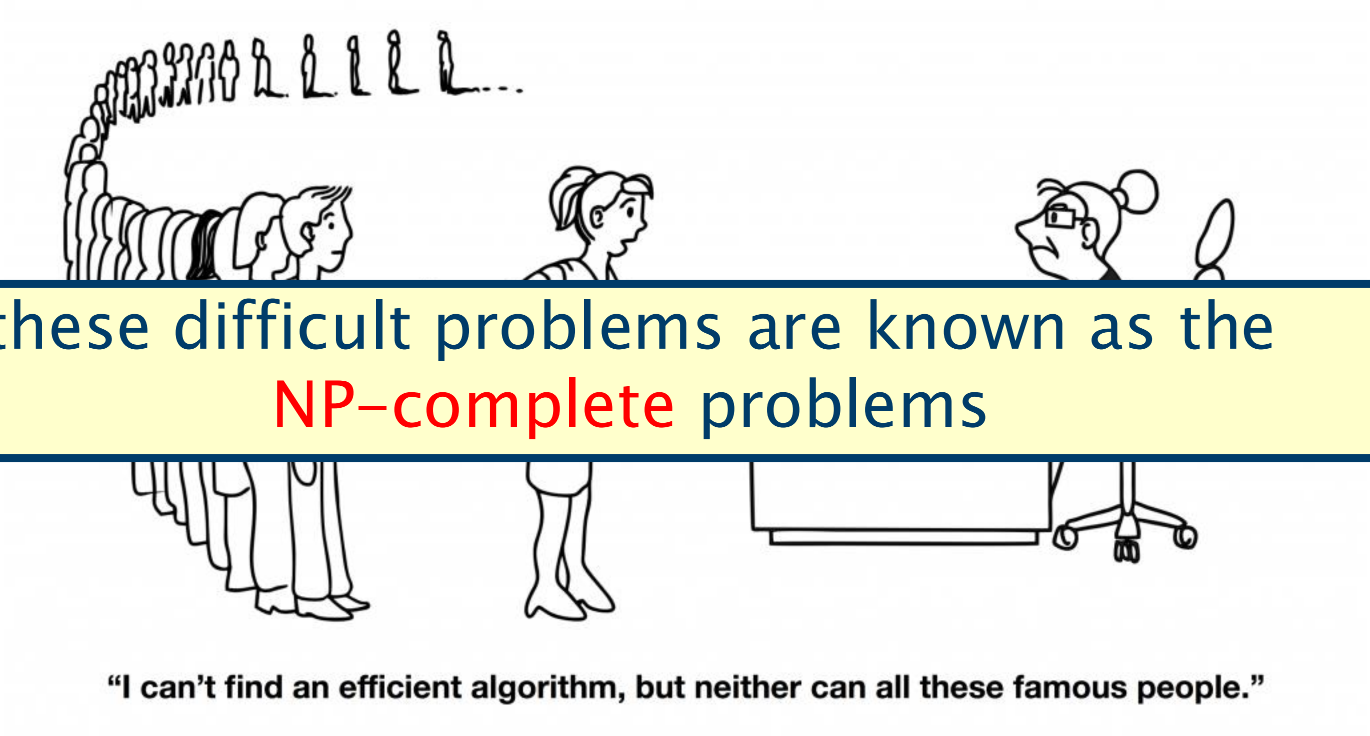


it can be very difficult to prove that a problem is intractable, and such proofs are rare

“I can’t find an efficient algorithm, because no such algorithm is possible!”

A brief interlude

You could try to prove that the Hamiltonian cycle problem is "just as hard" as a whole family of other difficult problems



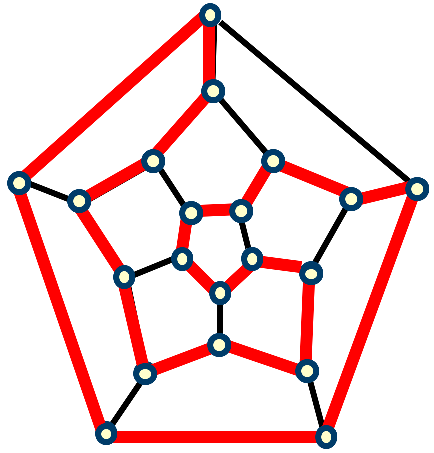
these difficult problems are known as the NP-complete problems

"I can't find an efficient algorithm, but neither can all these famous people."

A brief interlude

State of the Art for Hamiltonian cycle

- no polynomial-time algorithm has been found
- similarly, no proof of intractability has been found
- the problem is known to be an NP-complete problem



So what can we do in these circumstances?

- search for a polynomial-time algorithm should be given a lower priority
- could try to solve only “special cases” of the problem
- could look for an exponential-time algorithm that does reasonably well in practice
- could search for a polynomial-time algorithm that meets only some of the problem specifications

Section 4 – NP–completeness

Introduction (examples and discussion)

NP–complete problems

The classes P and NP

Polynomial–time reductions

Formal definition of NP–completeness

How to prove a problem is NP–complete

NP-complete problems

No polynomial-time algorithm is known for an NP-complete problem

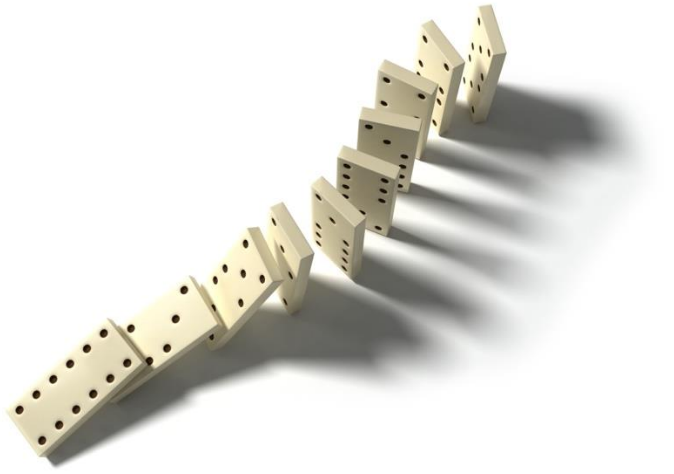
- however, if one of them is solvable in polynomial time, then they all are

No proof of intractability is known for an NP-complete problem

- however, if one of them is intractable, then they all are

There is a strong belief in the community that NP-complete problems are intractable

- we can think of all of them as being of equally difficulty



Intractable problems

Two different causes of intractability (no polynomial algorithm):

1. polynomial time is not sufficient in order to discover a solution
2. solution itself is so large that exponential time is needed to output it

We will be concerned with case 1

- there are intractability proofs for case 1
- some problems have been shown to be undecidable

  i.e. no algorithm of any sort could solve them (examples later)
- some decidable problems have been shown to be intractable

  i.e. can be solved but not in polynomial time

Example of case 2:

- consider problem of generating all cycles for a given graph

Intractable problems – Roadblock

A decidable problem that is intractable: Roadblock

- there are two players: A and B
- there is a network of roads, comprising intersections connected by roads
- each road is coloured either **black**, **blue** or **green**
- some intersections are marked either "**`A wins`**" or "**`B wins`**"
- a player has a fleet of cars located at intersections
  - at most one per intersection

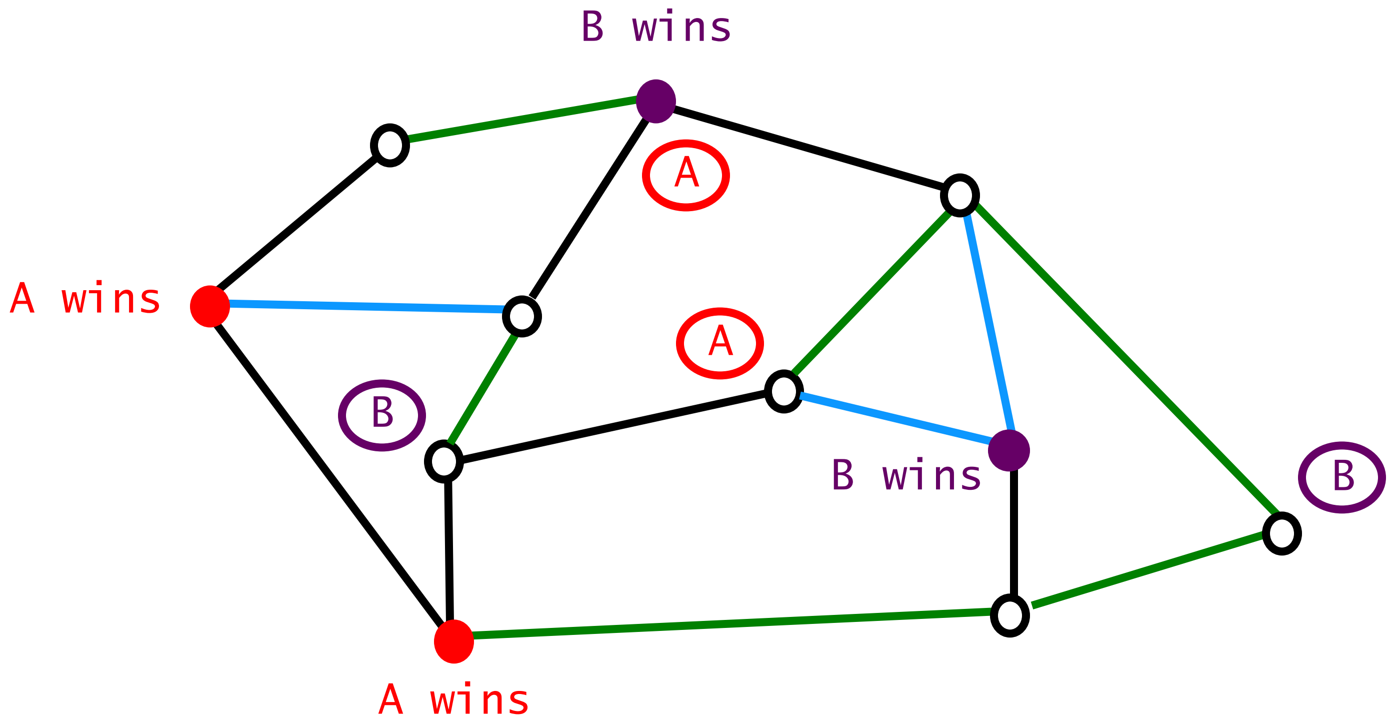
Player A begins, and subsequently players make moves in turn

- by moving one of their cars on one or more roads of the same colour
- a car may not stop at or pass over an intersection which already has a car

The problem is to decide, for a given starting configuration, whether A can win, regardless of what moves B takes

Intractable problems – Roadblock – Example

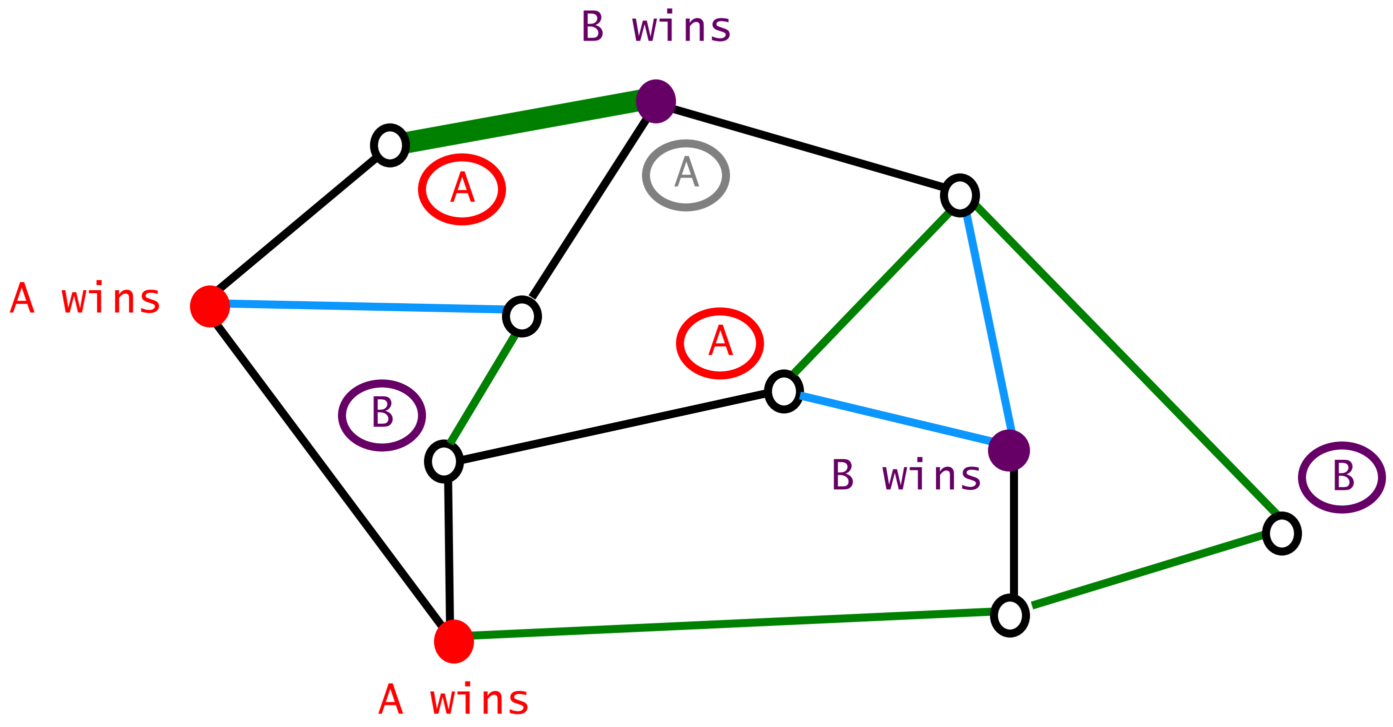
A moves first and A can win, no matter what B does. How?



Intractable problems – Roadblock – Example

A moves first and A can win, no matter what B does. How?

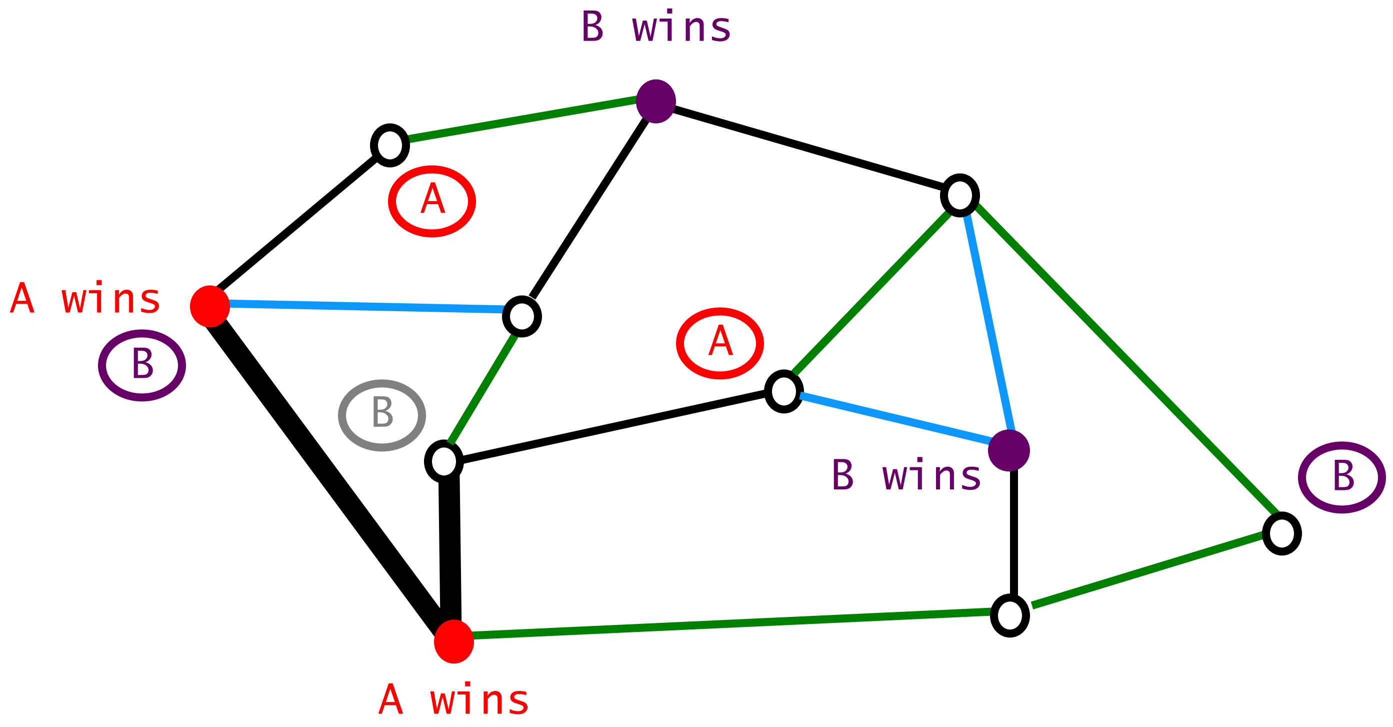
- A moves (along the **green** road)



Intractable problems – Roadblock – Example

A moves first and A can win, no matter what B does. How?

- A moves (along the **green** road)
- B moves (along the **black** road) to try and stop A from winning on its next turn

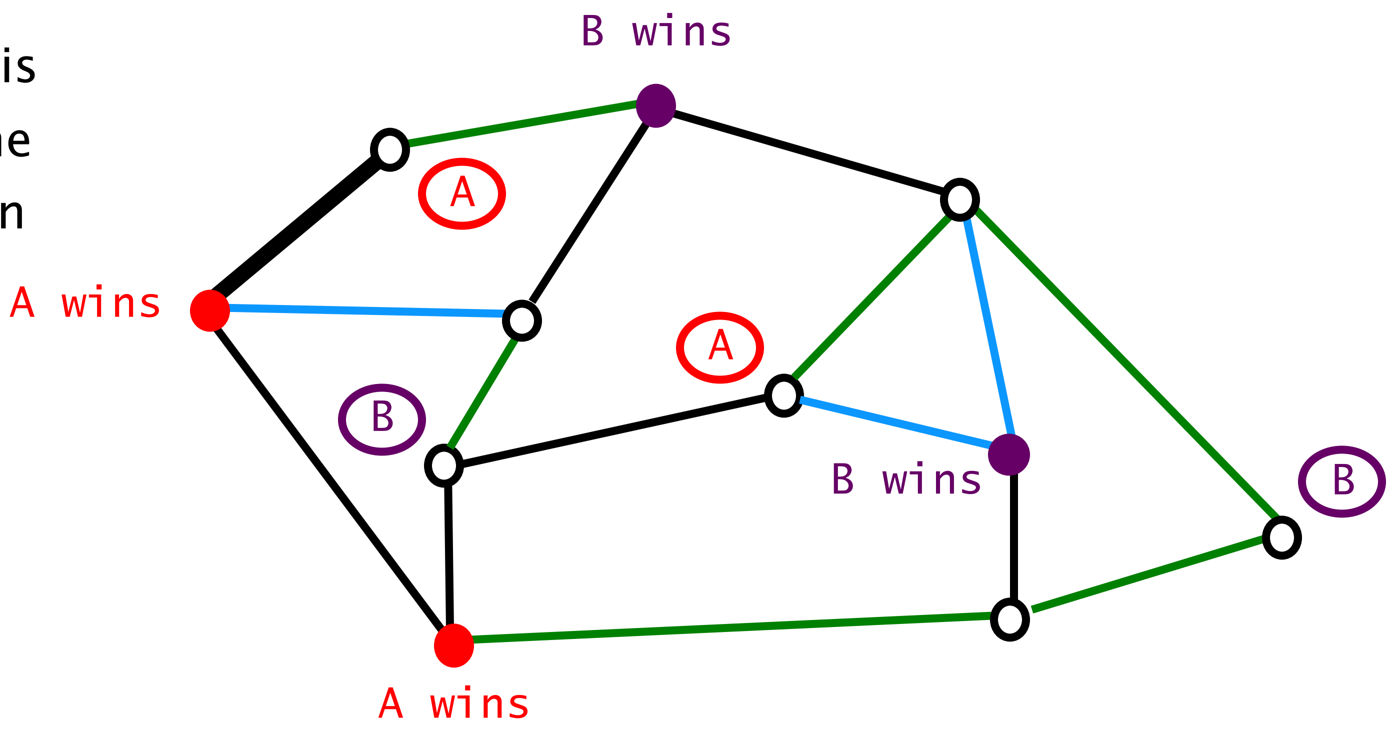


Intractable problems – Roadblock – Example

A moves first and A can win, no matter what B does. How?

- A moves (along the **green** road)
- B moves (along the **black** road) to try and stop A from winning on its next turn

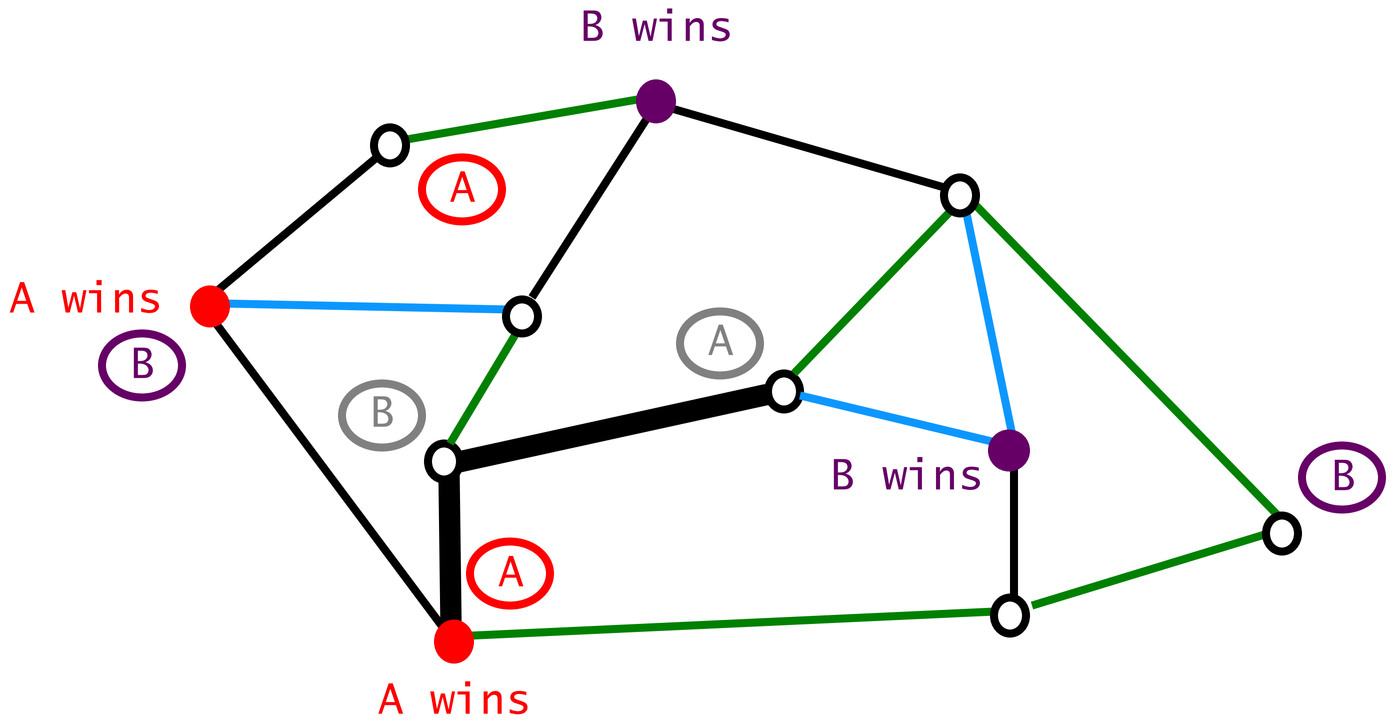
if B does not do this
A could move to the
same place and win



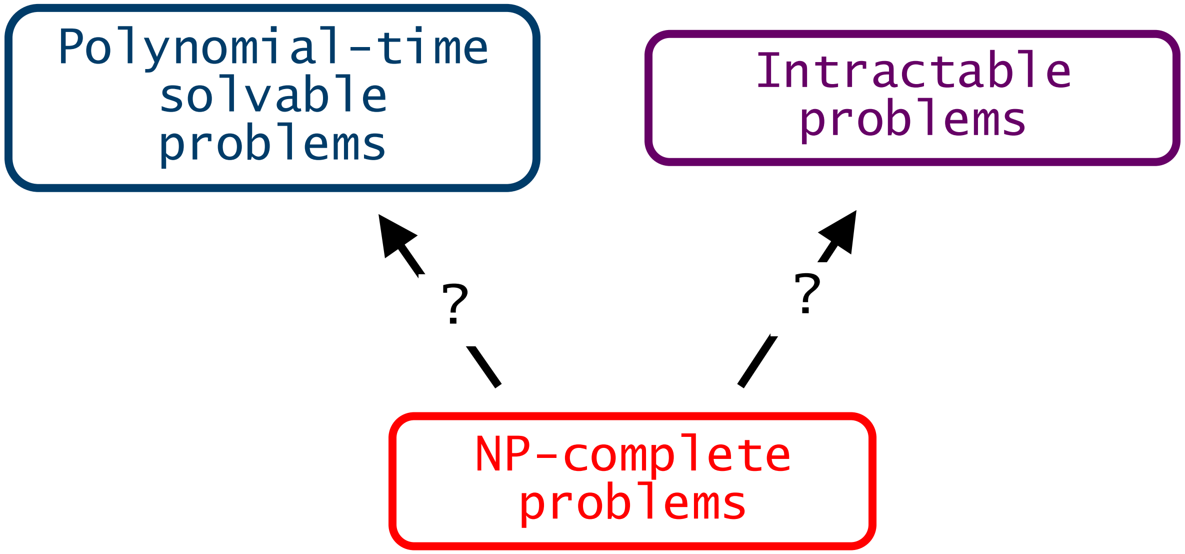
Intractable problems – Roadblock – Example

A moves first and A can win, no matter what B does. How?

- A moves (along the **green** road)
- B moves (along the **black** road) to try and stop A from winning
- but A can still win (by moving along the **black** road)



Summary



One of the question marks must be an 'equals' sign, while the other must be a 'not-equals' sign

Section 4 – NP-completeness

The classes P and NP

- P is the class of all decision problems that can be solved in polynomial time
- NP is the class of decisions problems solvable in non-deterministic polynomial time
  - the algorithms is allowed to guess (so when run can give different answers)
  - hence is apparently more powerful than a normal deterministic algorithm

- P is certainly contained within NP
  - a deterministic algorithm is just a special case of a non-deterministic one
  - but there is no problem in NP that has been proved not to be in P
  - the relationship between P and NP is the most notorious unsolved question in computing science
  - there is a million dollar prize if you can solve this question
    - a Clay Mathematics Institute Millennium Prize Problem

Problem and problem instances

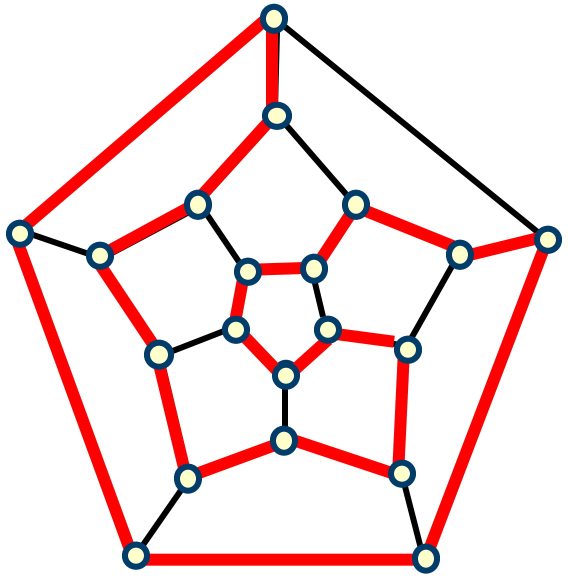
A **problem** is usually characterised by (unspecified) parameters

- typically there are infinitely many instances for a given problem

A **problem instance** is created by giving these parameters values

An **NP-complete** problem:

- Name: Hamiltonian Cycle (HC)
- Instance: a graph G
- Question: does G contain a cycle that visits each vertex exactly once?



This is an example of a **decision problem**

- the answer is ’yes’ or ’no’
- every instance is either a ’yes’-instance or a ’no’-instance

Other NP-complete problems

Name: Travelling Salesperson Decision Problem (TSDP)

Instance: a set of n cities and integer distance $d(i,j)$ between each pair of cities i, j, and a target integer K

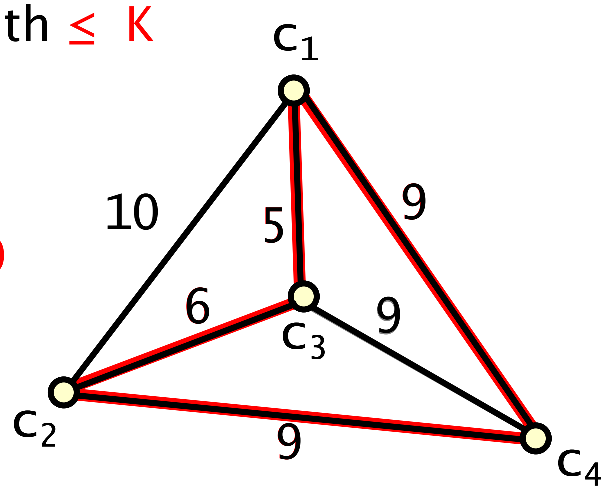
Question: is there a permutation $p_1p_2\ldots p_{n-1}p_n$ of $1,2,\ldots,n$ such that

$$d(p_1,p_2) + d(p_2,p_3) + \cdots + d(p_{n-1},p_n) + d(p_n,p_1) \leq K\ ?$$

- i.e. is there a 'travelling salesperson tour' of length $\leq K$

Example:

- there is a travelling salesperson tour of length 29
  - $d(1,3)+d(3,2)+d(2,4)+d(4,1)=5+6+9+9=29$
- there is no tour of length < 29



The travelling salesperson decision problem is NP-complete

Other NP–complete problems

Name: Clique Problem (CP)

Instance: a graph **G** and a target integer **K**

Question: does **G** contain a clique of size **K**?

- i.e. a set of K vertices for which there is an edge between all pairs

Other NP-complete problems

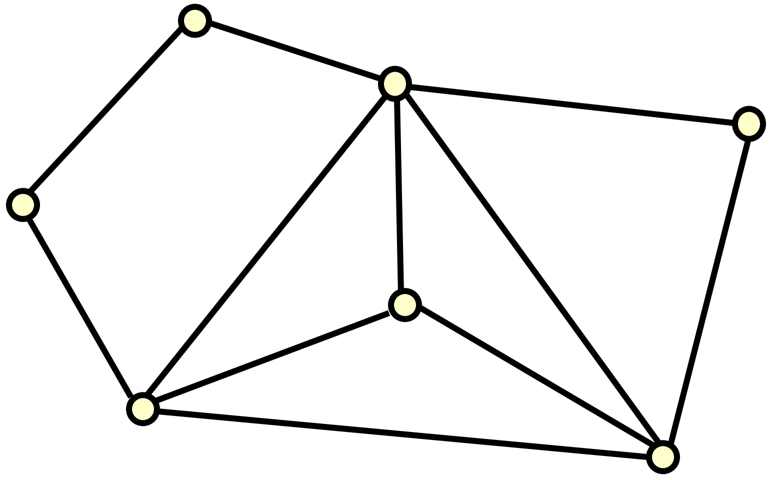
Name: Clique Problem (CP)

Instance: a graph **G** and a target integer **K**

Question: does **G** contain a clique of size **K**?

- i.e. a set of K vertices for which there is an edge between all pairs

Example:



Other NP-complete problems

Name: Clique Problem (CP)

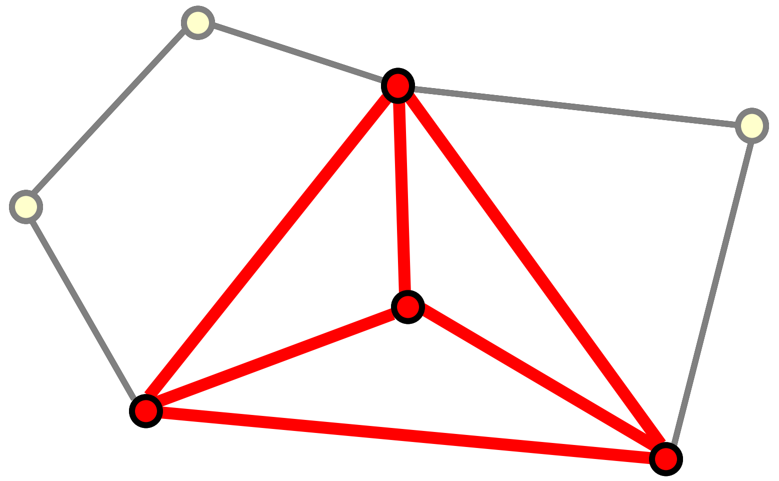
Instance: a graph G and a target integer K

Question: does G contain a clique of size K?

- i.e. a set of K vertices for which there is an edge between all pairs

Example:

- there is a clique of size 4
- there is no clique of size 5



The clique decision problem is NP-complete

Other NP-complete problems

Name: Graph Colouring Problem (GCP)

Instance: a graph G and a target integer K

Question: can one of K colours be attached to each vertex of G so that adjacent vertices always have different colours?

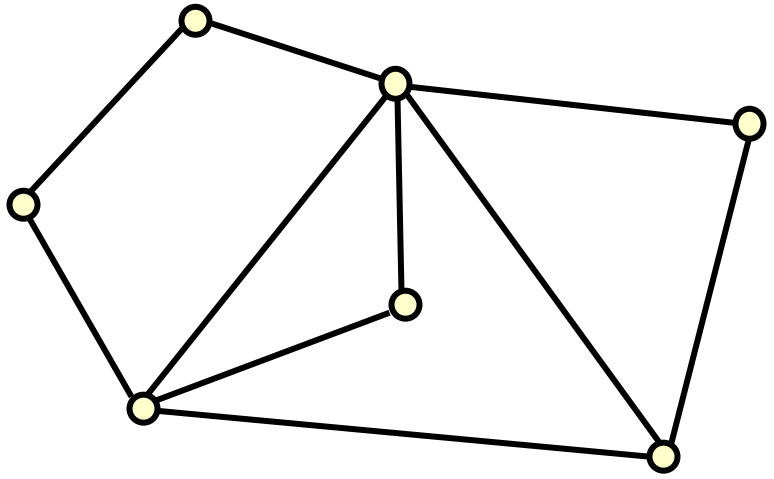
Other NP–complete problems

Name: Graph Colouring Problem (GCP)

Instance: a graph **G** and a target integer **K**

Question: can one of **K** colours be attached to each vertex of **G** so that adjacent vertices always have different colours?

Example:



Other NP-complete problems

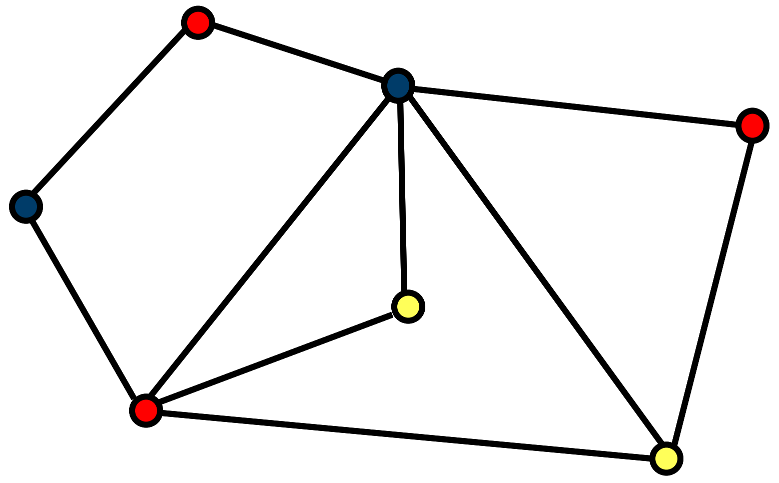
Name: Graph Colouring Problem (GCP)

Instance: a graph G and a target integer K

Question: can one of K colours be attached to each vertex of G so that adjacent vertices always have different colours?

Example:

- there is a colouring using 3 colours
- there is no colouring using 2 colours



The graph colouring decision problem is NP-complete

Other NP-complete problems

Name: Satisfiability (SAT)

Instance: Boolean expression **B** in **conjunctive normal form (CNF)**

- CNF: $C_1 \land C_2 \land \dots \land C_n$ where each C_i is a clause
- Clause C: $(l_1 \lor l_2 \lor \dots \lor l_m)$ where each l_j is a literal
- Literal l: a variable x or its negation $\neg x$

Question: is **B satisfiable**?

- i.e. can values be assigned to the variables that make B true?

Example:

- $B = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land$
  $(x_2 \lor \neg x_3 \lor x_4)$

Other NP-complete problems

Name: Satisfiability (SAT)

Instance: Boolean expression **B** in **conjunctive normal form (CNF)**

- CNF: $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each C_i is a clause
- Clause C: $(l_1 \vee l_2 \vee \ldots \vee l_m)$ where each l_j is a literal
- Literal l: a variable x or its negation $\neg x$

Question: is **B satisfiable?**

- i.e. can values be assigned to the variables that make B true?

Example:

- $B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge$
  $(x_2 \vee \neg x_3 \vee x_4)$
- B is satisfiable: $x_1=$**true**, $x_2=$**false**, $x_3=$**true**, $x_4=$**true**

The satisfiability problem is NP-complete

Optimisation and search problems

An optimisation problem: find the maximum or minimum value

- e.g. the travelling salesperson optimisation problem (TSOP) is to find the minimum length of a tour

A search problem: find some appropriate optimal structure

- e.g. the travelling salesperson search problem (TSSP) is to find a minimum length tour

NP-completeness deals primarily with decision problems

- corresponding to each instance of an optimisation or search problem is a family of instances of a decision problem by setting 'target' values
- almost invariably, an optimisation or search problem can be solved in polynomial time if and only if the corresponding decision problem can (we will consider some examples of this in the tutorials)

Next lecture

Introduction (examples and discussion)

NP-complete problems

The classes P and NP

Polynomial-time reductions

Formal definition of NP-completeness

How to prove a problem is NP-complete