Algorithmics I

Section 4 – NP completeness

Dr. Gethin Norman

School of Computing Science
University of Glasgow

gethin.norman@glasgow.ac.uk

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:

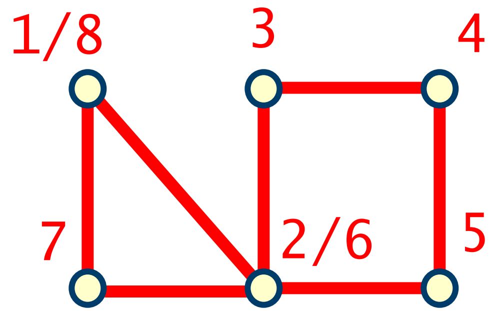
- searching a sorted list $O(\log n)$ (for an array/list of length n)
- finding the max value $O(n)$ (for an array/list of length n)
- sorting $O(n \log n)$ (for an array/list of length n)
- distance between two strings $O(n^2)$ (for two strings of length n)
- finding a shortest path $O(n^2)$ (for weighted 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

Recall the Eulerian cycle problem (AF2)

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

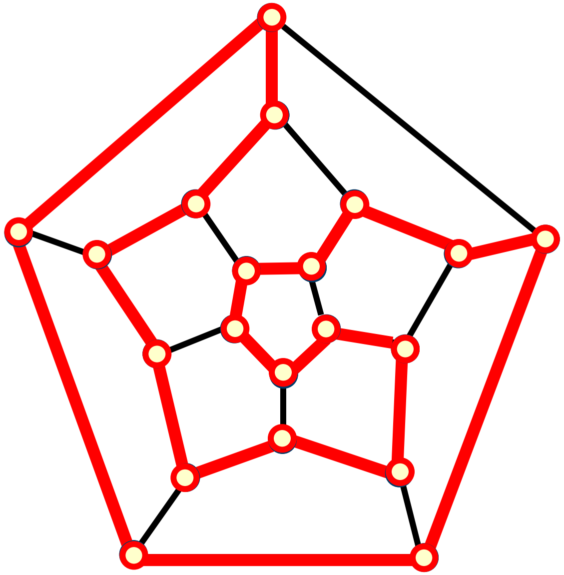
therefore we can test whether G has an Euler cycle (and find one) in:

- $O(n^2)$ time if G is represented by an adjacency matrix
- $O(m+n)$ time if G is represented by adjacency lists
- where $m=|E|$ and $n=|V|$

Recall the Hamiltonian cycle problem (AF2)

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

Recall the Hamiltonian cycle problem (AF2)

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

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
- what happens when computers become faster?

	current computer	computer 100 times faster	computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
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 to 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

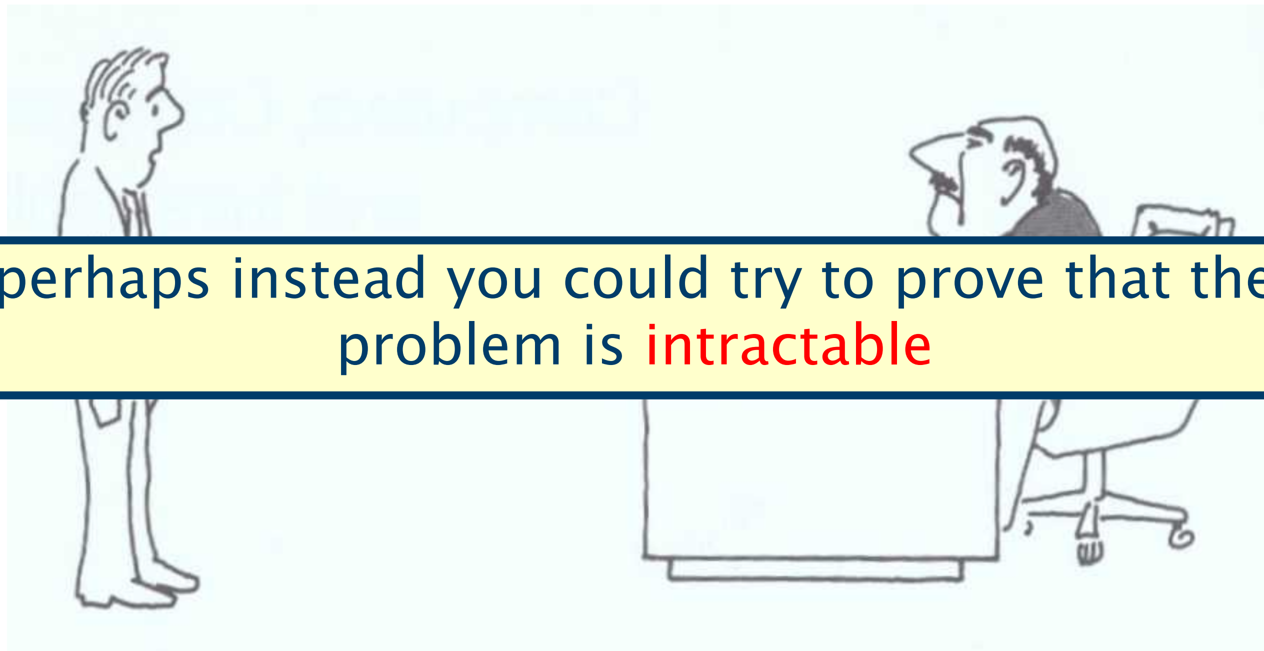
- often polynomial-time algorithms require some extra insight
- often exponential-time algorithms are variations on exhaustive search

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:



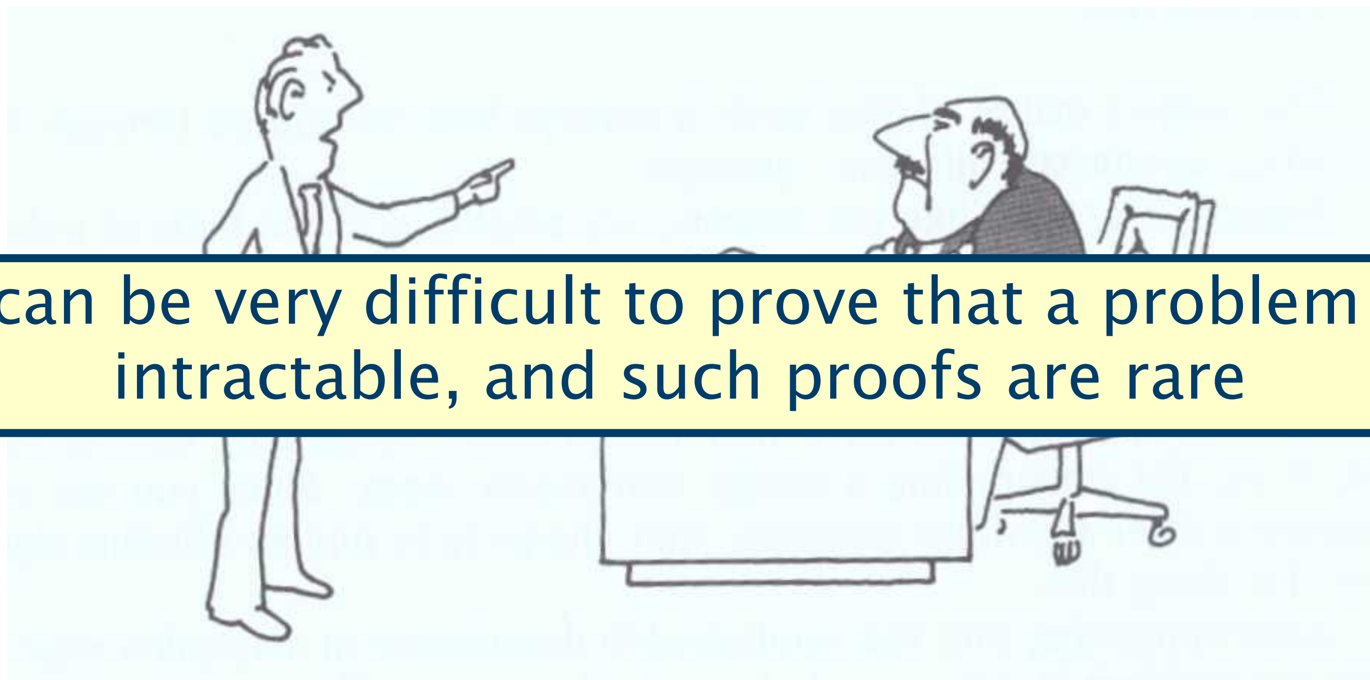
perhaps instead you could try to prove that the problem is intractable

“I cannot find an efficient algorithm I guess I’m too dumb”

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

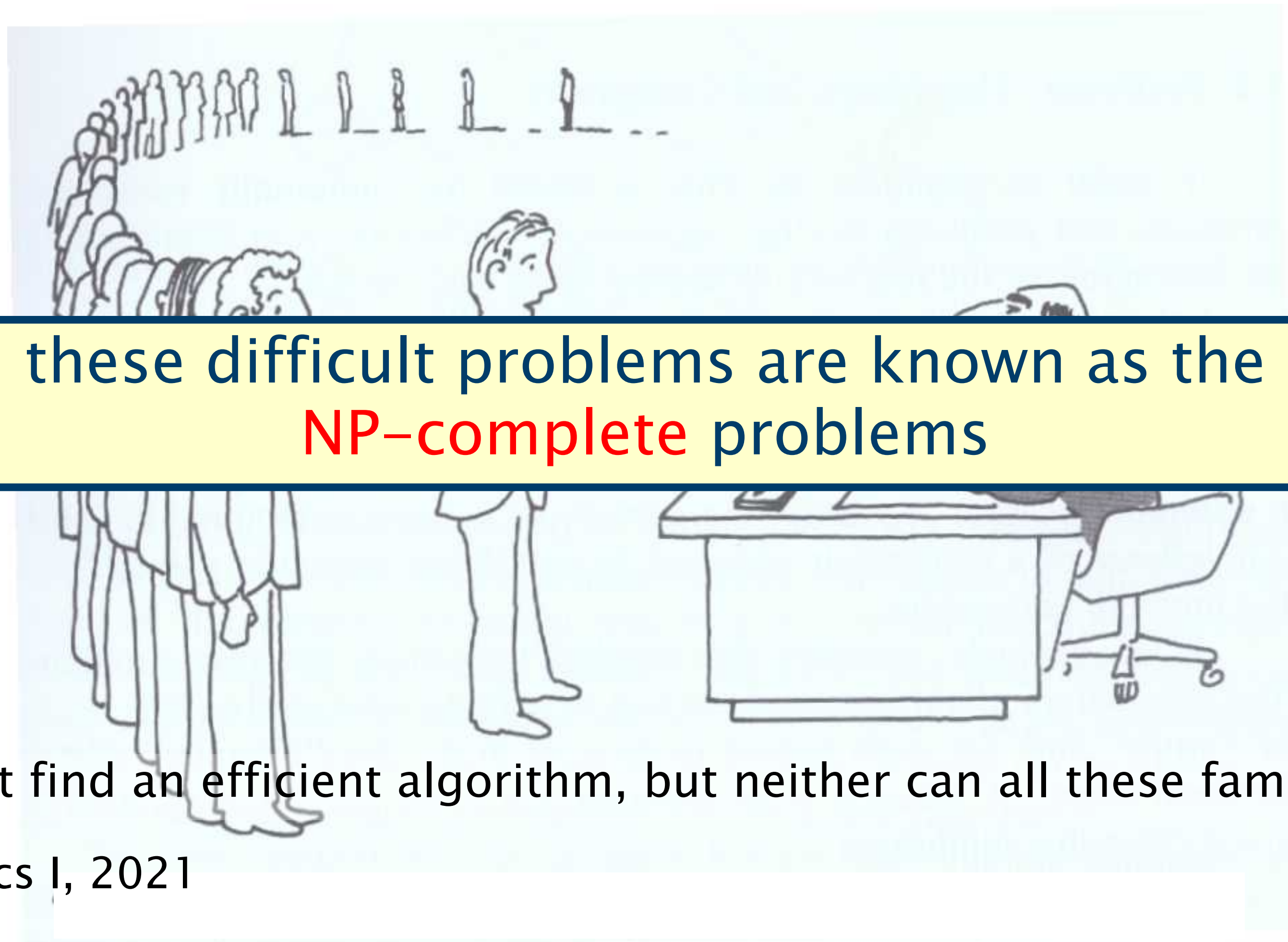


it can be very difficult to prove that a problem is intractable, and such proofs are rare

“I cannot 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

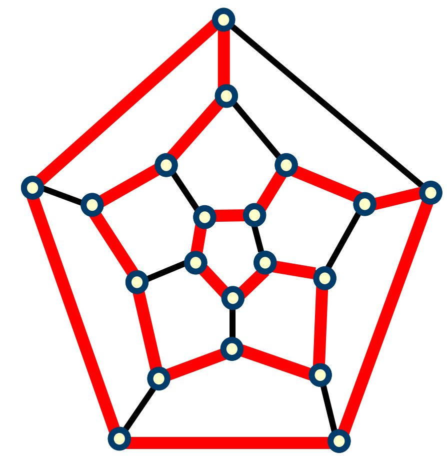


these difficult problems are known as the NP-complete problems

"I cannot 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

NP-complete problems

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

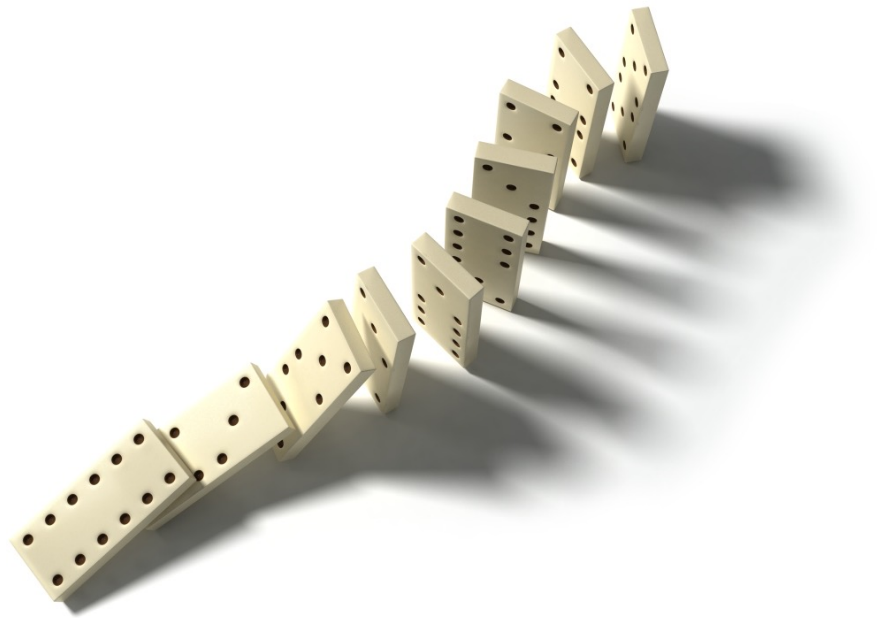
- however, if one of them is solvable in polynomial time, then they all are

No proof of intractability is known for a 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 equivalent 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

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

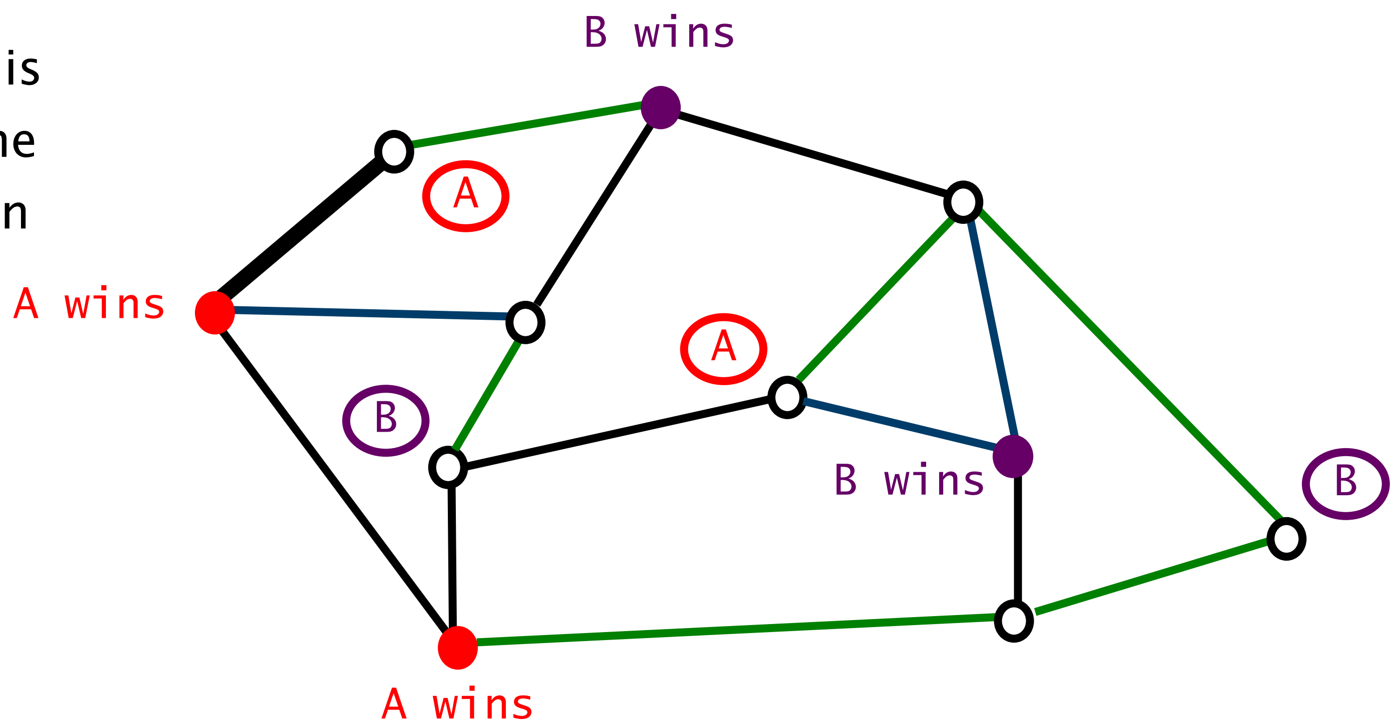
The problem is to decide, for a given starting configuration, whether A can win, regardless of what moves B takes

Intractable problems – Roadblock – Example

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

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

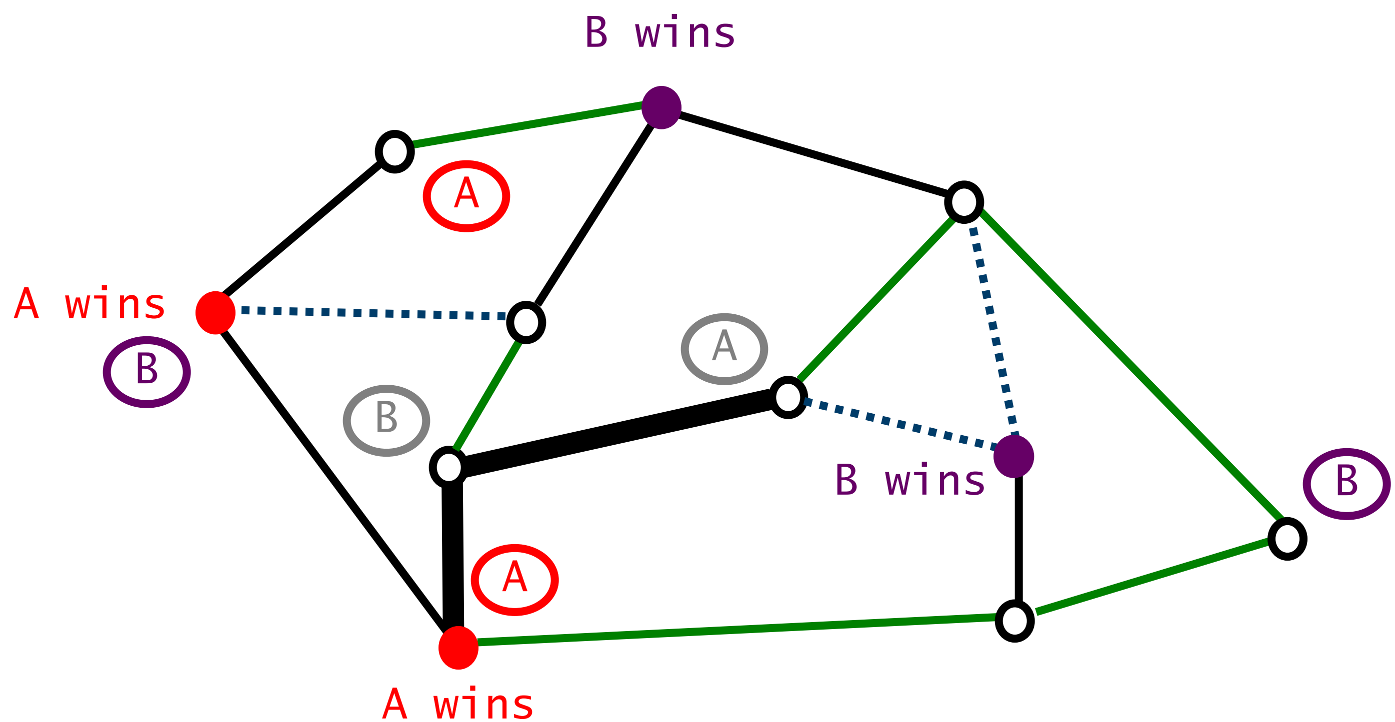
if B does not do this
A could move to the
same place and win



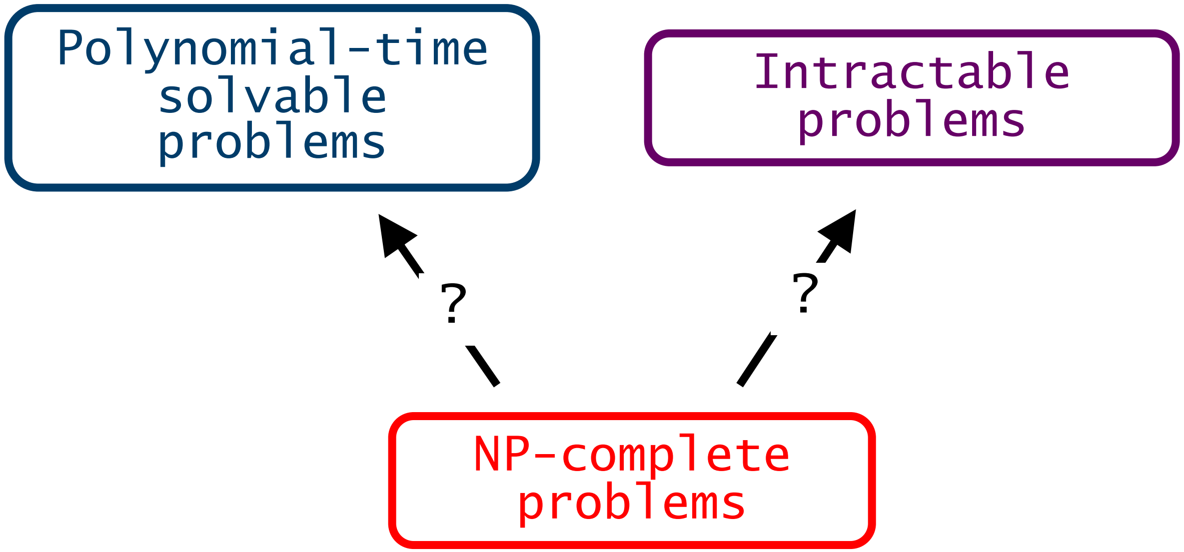
Intractable problems – Roadblock – Example

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

Problem and problem instances

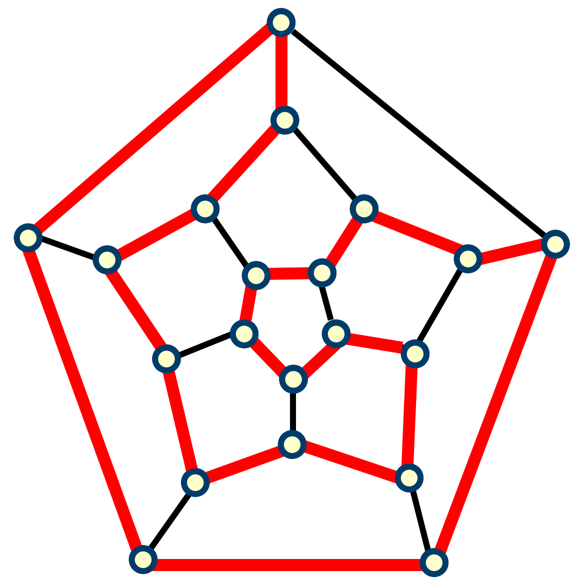
A problem is usually characterised by (unspecified) parameters

- 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

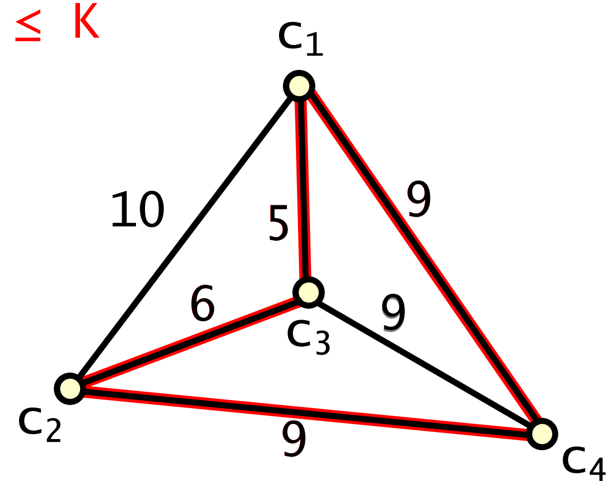
Name: Travelling Salesman Decision Problem (TSDP)

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 $p_1 p_2 \ldots p_{n-1} p_n$ of $1,2,\ldots,n$ such that

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

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



Example:

- there is a travelling salesman 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 salesman decision problem is NP-complete

Other NP-complete problems

Name: Clique Problem (CP)

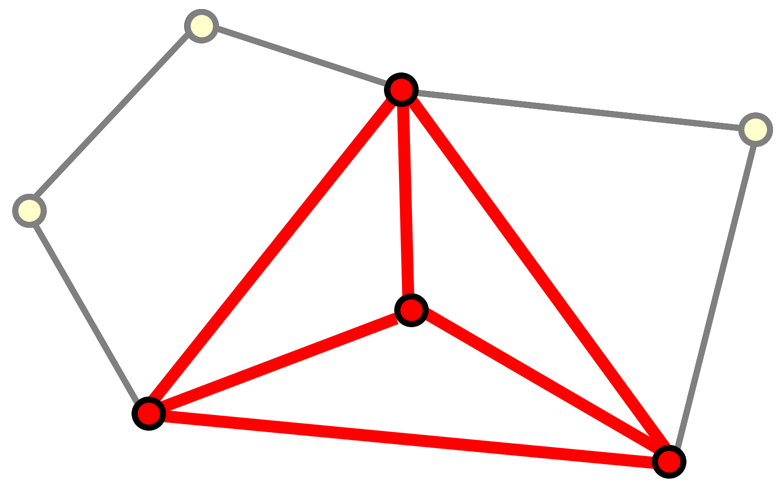
Instance: a graph G and a target integer K

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

Example:

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



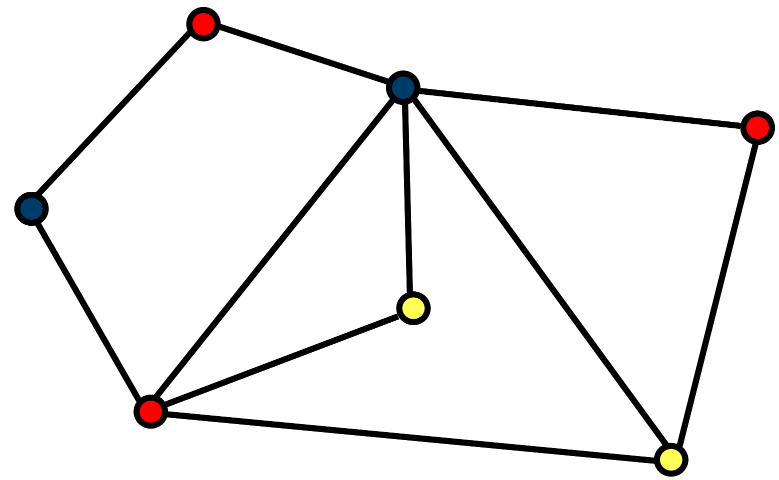
The clique decision problem is NP-complete

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?

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: $C_1 \wedge C_2 \wedge \dots \wedge C_n$ where each C_i is a clause
- Clause C: $(l_1 \vee l_2 \vee \dots \vee l_m)$ 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 salesman optimisation problem (TSOP) is to find the minimum length of a tour

A search problem: find some appropriate optimal structure

- e.g. the travelling salesman 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)

The class P

P is the class of all decision problems that can be solved in polynomial time

Fortunately, many problems are in P

- is there a path of length ≤K from vertex u to vertex v in a graph G?
- is there a spanning tree of weight ≤K in a graph G?
- is a graph G bipartite?
- is a graph G connected?
- deadlock detection: does a directed graph D contain a cycle?
- text searching: does a text t contain an occurrence of a string s?
- string distance: is d(s,t)≤K for strings s and t?
- …

P often extended to include search and optimisation problems

The class NP

The decision problems solvable in non-deterministic polynomial time

- a non-deterministic algorithm can make non-deterministic choices
  - the algorithm 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 is that containment strict?

- there is no problem known to be in NP and known 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

Non-deterministic algorithms (NDAs)

Such an algorithm has an extra operation: non-deterministic choice

```
int nonDeterministicChoice(int n)
// returns a positive integer chosen from the range 1,…,n
```

- an NDA has many possible executions depending on values returned

An NDA "solves" a decision problem Π if

- for a 'yes'-instance I of Π there is some execution that returns 'yes'
- for a 'no'-instance I of Π there is no execution that returns 'yes'

and "solves" a decision problem Π in polynomial time if

- for every 'yes'-instance I of Π there is some execution that returns 'yes' which uses a number of steps bounded by a polynomial in the input
- for a 'no'-instance I of Π there is no execution that returns 'yes'

Non-deterministic algorithms (NDAs)

An NDA "solves" a decision problem π if

- for a 'yes'-instance I of π there is some execution that returns 'yes'
- for a 'no'-instance I of π there is no execution that returns 'yes'

Clearly such algorithms are not useful in practice

- who would use an algorithm that sometimes gives the right answer

However they are a useful mathematical concept for defining the classes of NP and NP-complete problems

Non-deterministic algorithms – Example

Graph colouring

```
 // return true if graph g is k-colourable and false otherwise
boolean nDGC(Graph g, int k){

 for (each vertex v in g) v.setColour(nonDeterministicChoice(k));

 for (each edge {u,v} in g)
  if (u.getColour() == v.getColour()) return false;
 return true;
}
```

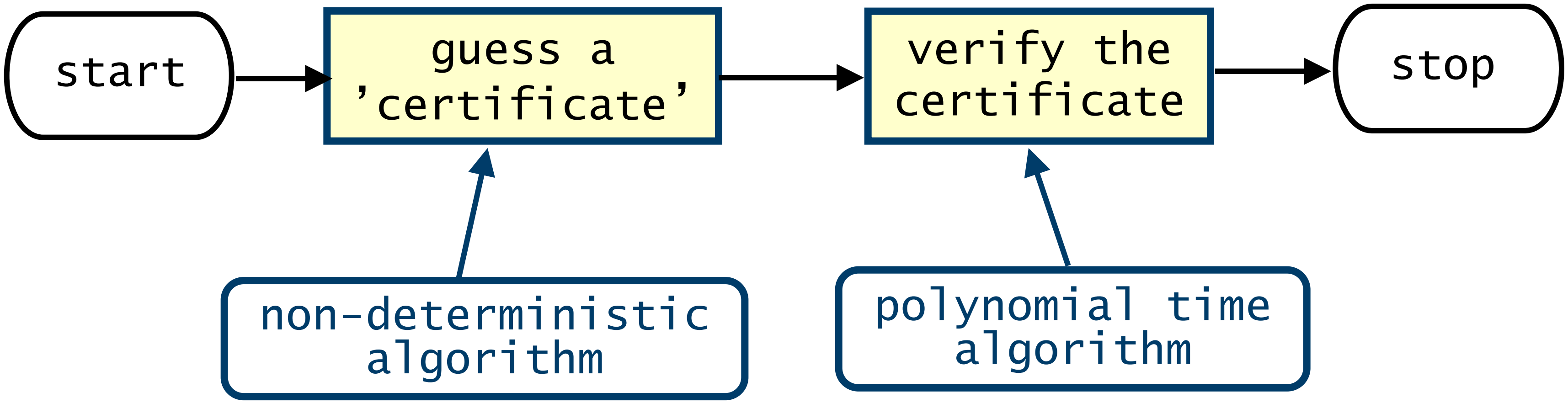
"verify" the colouring

"guess" a colour for each vertex

Non-deterministic algorithms

An non-deterministic algorithm can be viewed as

- a guessing stage (non-deterministic)
- a checking stage (deterministic and polynomial time)



Polynomial time reductions

A **polynomial-time reduction (PTR)** is a mapping f from a decision problem Π_1 to a decision problem Π_2 such that:

for every instance I_1 of Π_1 we have

- the instance $f(I_1)$ of Π_2 can be constructed in polynomial time
- $f(I_1)$ is a 'yes'-instance of Π_2 if and only if I_1 is a 'yes'-instance of Π_1

We write $\Pi_1 \propto \Pi_2$ as an abbreviation for:

there is a polynomial-time reduction from Π_1 to Π_2

Polynomial time reductions – Properties

Transitivity: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ implies that $\Pi_1 \propto \Pi_3$

Since $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ we have

- a PTR f from Π_1 to Π_2
- a PTR g from Π_2 to Π_3

Now for any instance I_1 of Π_1 since f is PTR we have

- $I_2=f(I_1)$ is an instance of Π_2 that can be constructed in polynomial time
- I_2 has the same answer as I_1

and since g is a PTR we have

- $I_3=g(I_2)$ is an instance of Π_3 that can be constructed in polynomial time
- I_3 has the same answer as I_2

Polynomial time reductions – Properties

Transitivity: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ implies that $\Pi_1 \propto \Pi_3$

Since $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ we have

- a PTR f from Π_1 to Π_2
- a PTR g from Π_2 to Π_3

Putting the results together: for any instance I_1 of Π_1

- $I_3=g(f(I_1))$ is an instance of Π_3 constructed in polynomial time
- I_3 has the same answer as I_1
- i.e. the composition of f and g is a PTR from from Π_1 to Π_3

Polynomial time reductions – Properties

Relevance to P: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \in \mathbf{P}$ implies that $\Pi_1 \in \mathbf{P}$

- to solve an instance of Π_1, reduce it to an instance of Π_2
- roughly speaking, $\Pi_1 \propto \Pi_2$ means that Π_1 is 'no harder' than Π_2

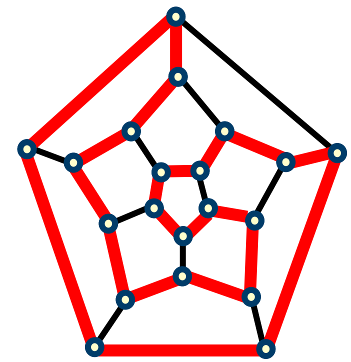
  i.e. if we can solve Π_2, then we can solve Π_1 without much more effort

  - just need to additional perform a polynomial time reduction

Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesman problem

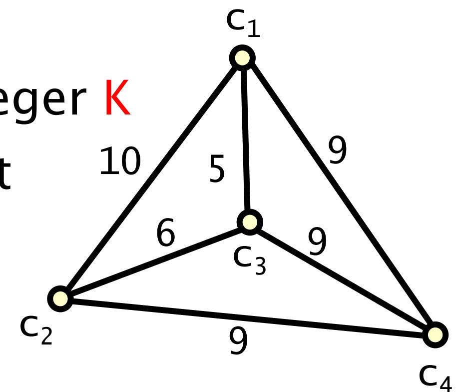
Hamiltonian Cycle Problem (HC)



- instance: a graph G
- question: does G contain a cycle that visits each vertex exactly once?

Travelling Salesman Decision Problem (TSDP)

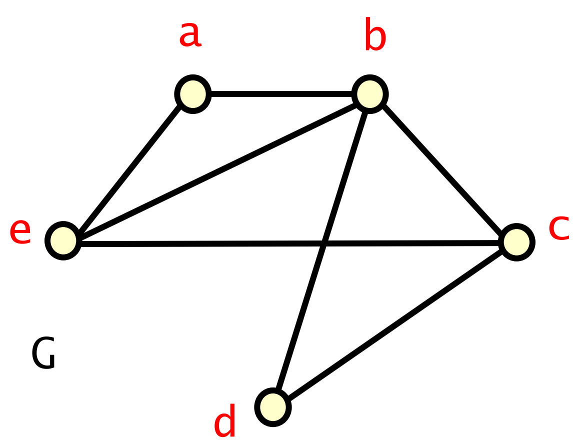
- instance: a set of n cities and integer distance $d(i,j)$ between each pair of cities i,j, and a target integer K
- question: is there a permutation p of $\{1,2,\ldots,n\}$ such that $d(p_1,p_2)+d(p_2,p_3)+\cdots+d(p_{n-1},p_n)+d(p_n,p_1)\leq K$?
  - i.e. is there a 'travelling salesman tour' of length $\leq K$

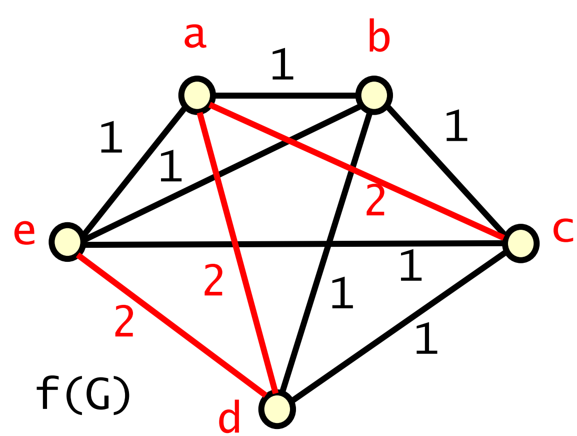


Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesman problem

- G = (V,E) is an instance of HC
- construct TSDP instance f(G) where
  - cities = V
  - d(u,v)=1 if {u,v}∈E and 2 otherwise (is not an edge of G)
  - K = |V|

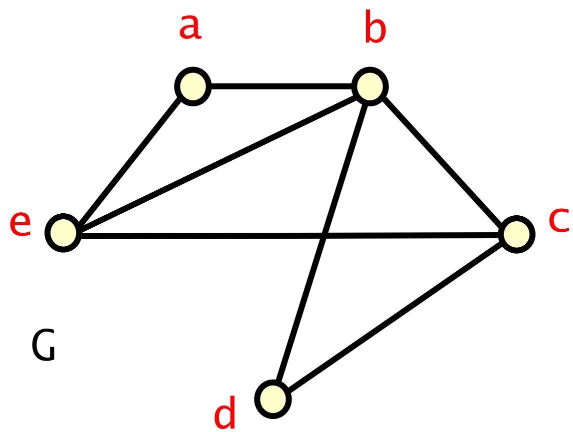


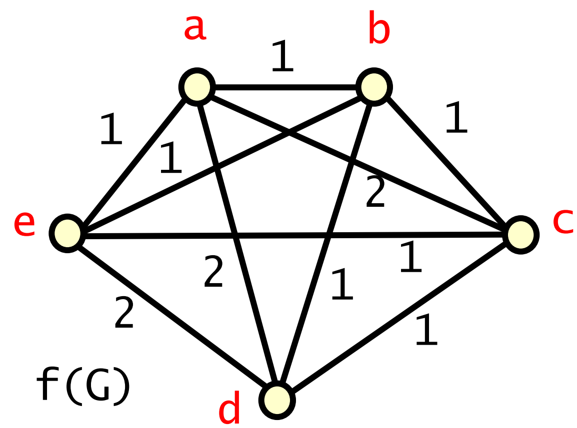


Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesman problem

- G = (V,E) is an instance of HC
- construct TSDP instance f(G)





- f(G) can be constructed in polynomial time
- f(G) has a tour of length ≤|V| if and only if G has a Hamiltonian cycle (tour includes |V| edges so cannot take any of the edges with weight 2)
- therefore TSDP∈P implies that HC∈P
- equivalently HC∉P implies that TSDP∉P (contrapositive)

NP-completeness

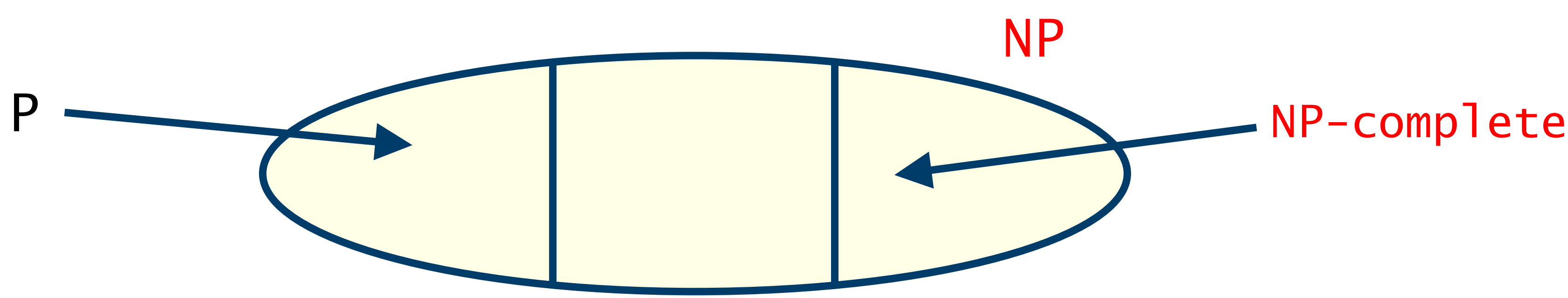
A decision problem π is NP-complete if

1. $\pi \in NP$
2. for every problem π' in NP: π' is polynomial-time reducable to π

Consequences of definition

- if π is NP-complete and $\pi \in P$, then $P = NP$
- every problem in NP can be solved in polynomial time by reduction to Π
- supposing $P \neq NP$, if π is NP-complete, then $\pi \notin P$

The structure of NP if $P \neq NP$



Proving NP-completeness

A decision problem π is NP-complete if

1. $\pi \in NP$
2. for every problem π' in NP: π' is polynomial-time reducable to π

How can we possibly prove any problem to be NP-complete?

- it is not feasible to describe a reduction from every problem in NP
- however, suppose we knew just one NP-complete problem π_1

To prove π_2 is NP-complete enough to show

- π_2 is in NP
- there exists a polynomial-time reduction from π_1 to π_2

Proving NP-completeness

A decision problem π is NP-complete if

1. $\pi \in NP$
2. for every problem π' in NP: π' is polynomial-time reducable to π

Suppose we knew just one NP-complete problem π_1, then to prove π_2 is NP-complete it is enough to show

- π_2 is in NP
- there exists a polynomial-time reduction from π_1 to π_2

Correctness of the approach

- for any $\pi \in NP$, since π_1 is NP-complete we have $\pi \propto \pi_1$
- since $\pi \propto \pi_1$, $\pi_1 \propto \pi_2$ and $\propto$ is transitive, it follows that $\pi \propto \pi_2$
- since $\pi \in NP$ was arbitrary, $\pi \propto \pi_2$ for all $\pi \in NP$
- and hence π_2 is NP-complete

Proving NP-completeness

The first NP-complete problem?

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**

Proving NP-completeness

The first NP-complete problem?

Cook's Theorem (1971): Satisfiability (SAT) is NP-complete

- the proof consists of a generic polynomial-time reduction to SAT from an abstract definition of a general problem in the class NP
- the generic reduction could be instantiated to give an actual reduction for each individual NP problem

Given Cook's theorem, to prove a decision problem Π is NP-complete it is sufficient to show that:

- Π is in NP
- there exists a polynomial-time reduction from SAT to Π

Clique is NP-complete

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

To prove Clique is NP -complete

- show CP is in NP (straightforward)
- there exists a polynomial-time reduction from SAT to CP

Clique is NP-complete

To complete the proof we need to show SAT $\propto$ CP

- i.e. a polynomial time reduction from SAT to CP

This is not examinable – this is just to show you that it is possible to build PTRs between very different problems

Clique is NP-complete

To complete the proof we need to show SAT $\propto$ CP

- i.e. a polynomial time reduction from SAT to CP

Given an instance B of SAT we construct (G,K) an instance of CP

- K number of clauses of B
- vertices of G are pairs (l,C) where l is a literal in clause C
- {(l,C),(m,D)} is an edge of G if and only if $l \neq \neg m$ and $C \neq D$
  - recall that $\neg(\neg x)=x$ so $l \neq \neg m$ is equivalent to $\neg l \neq m$
  - edge if distinct literals from different clauses can be satisfied simultaneously
- polynomial time construction ($O(n^2)$ where n is the number of literals)
  - worst case: to construct edges we need to compare every literal with every other literal

This is a polynomial time reduction since:

- B has a satisfying assignment if and only if G has a clique of size K

Clique is NP-complete

To prove it is a polynomial time reduction we can show:

If B has a satisfying assignment, then

- if we choose a `true` literal in each clause the corresponding vertices form a clique of size K in G

If G has a clique of size K, then

- assigning each literal associated with a vertex in the clique to be `true` yields a satisfying assignment for B

Clique is NP-complete

Why does the construction work?

{(l,C),(m,D)} is an edge if and only if **$l \neq \neg m$** and **$C \neq D$**

- only edges between literals in distinct clauses
- only edges between literals that can be satisfied simultaneously

Therefore in a clique of size K (recall K is the number of clauses)

- must include one literal from each clause (i.e. from K clauses)
- we can satisfy all the literals in the clique simultaneously
- this means we can satisfy all clauses
  - a clause is a disjunction of literals and we can satisfy one of them
- and therefore satisfy B
  - B is the conjunction of the clauses

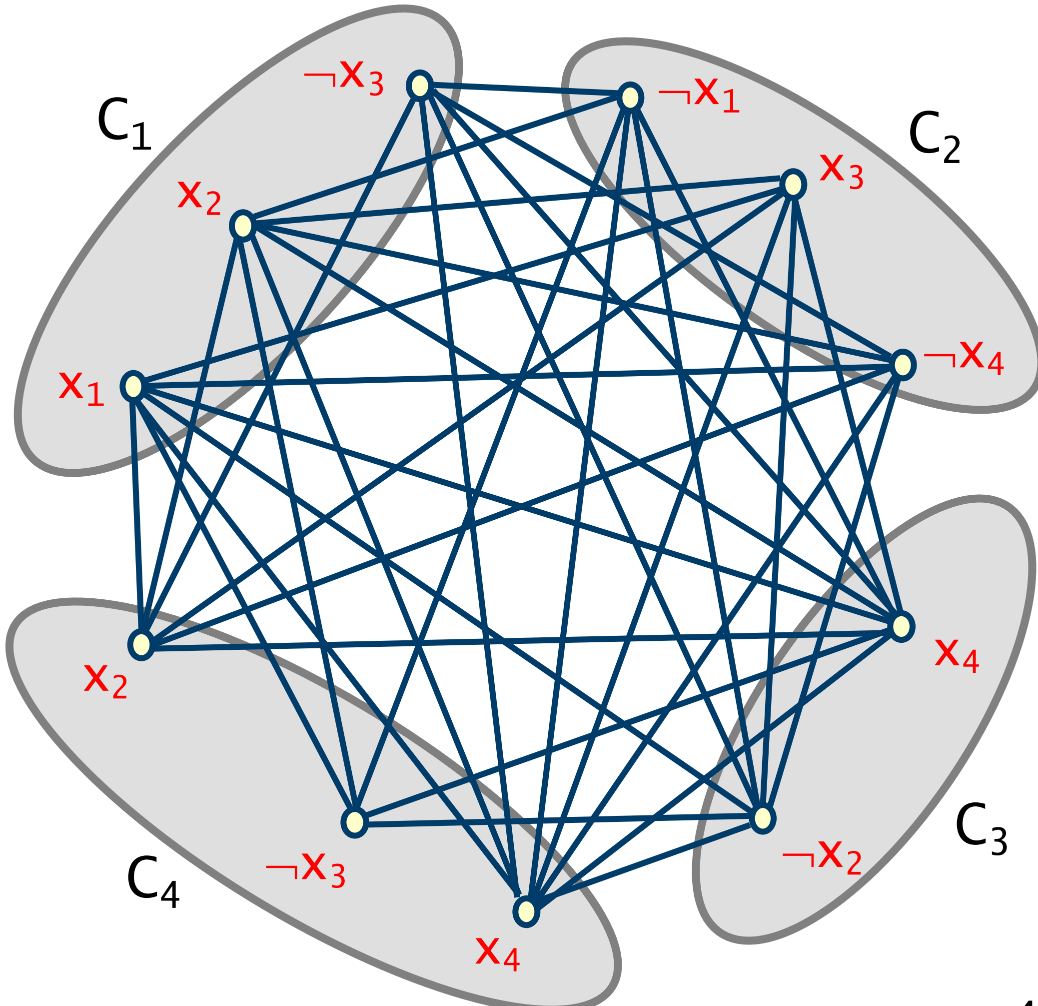
Clique is NP-complete – Example

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

– there are $K = 4$ clauses

The graph G

– vertices of G are pairs (l,C) where l is a literal in clause C
– {(l,C),(m,D)} is an edge if and only if $l \neq \neg m$ and $C \neq D$



Clique is NP-complete

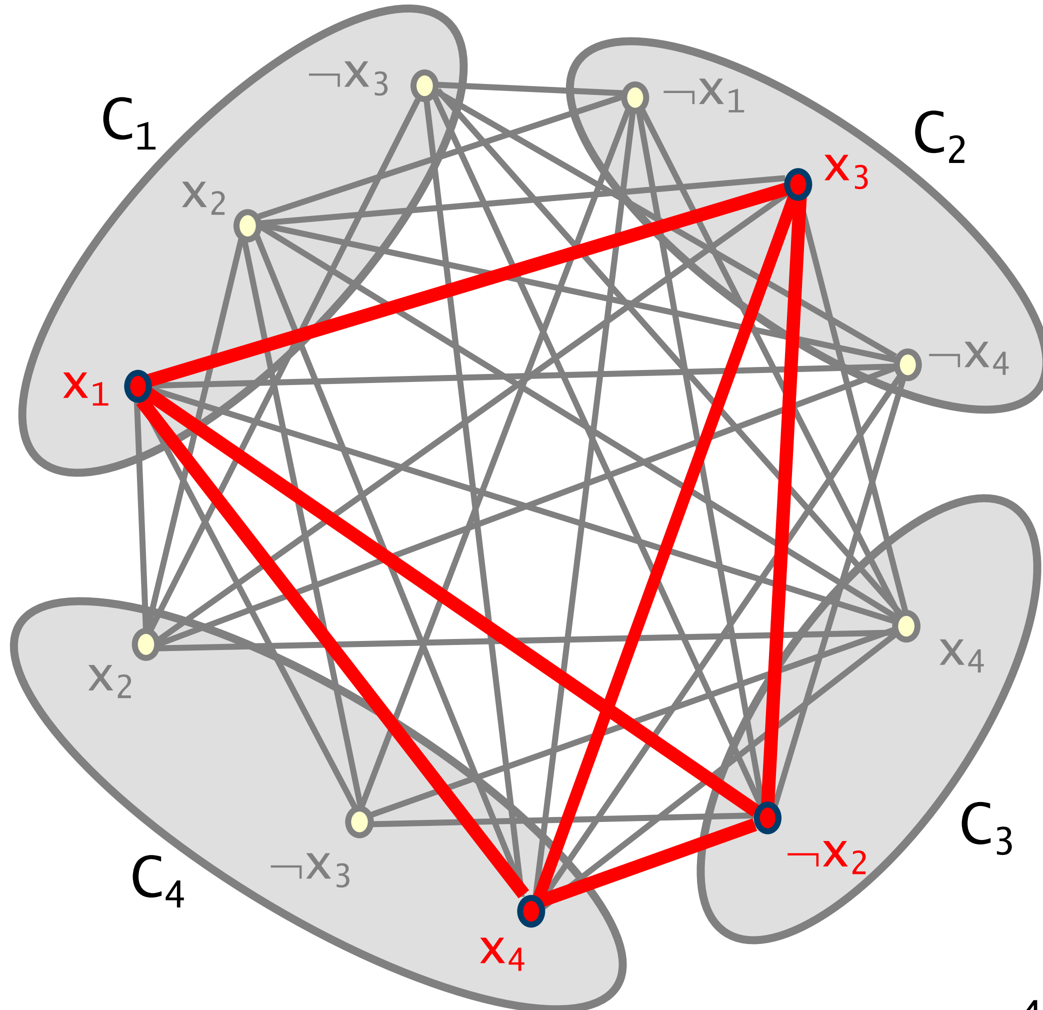
$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph **G**

G has a clique of size 4
if and only if
B has a satisfying assignment

satisfying assignment
clique of size 4



Problem restrictions

A restriction of a problem consists of a subset of the instances of the original problem

- if a restriction of a given decision problem Π is NP-complete, then so is Π
- given NP-complete problem Π, a restriction of Π might be NP-complete

For example a clique restricted to cubic graphs is in P

- (a cubic graph is a graph in which every vertex belongs to 3 edges)
- a largest clique has size at most 4 so exhaustive search is $O(n^4)$

While graph colouring restricted to cubic graphs is NP-complete

- not proved here

Problem restrictions

K-colouring

- restriction of Graph Colouring for for a fixed number K of colours
- 2-colouring is in P (it reduces to checking the graph is bipartite)
- 3-colouring is NP-complete

K-SAT

- restriction of SAT in which every clause contains exactly K literals
- 2-SAT is in P (proof is a tutorial exercise)
- 3-SAT is NP-complete
- showing 3-SAT $\in$ NP is easy we will just show SAT $\propto$ 3-SAT

SAT $\propto$ 3-SAT

Given instance **B** of SAT will construct an instance **B'** of 3-SAT

For each clause **C** of **B** we construct a number of clauses of **B'**

- if $C=l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'
- B' holds if and only if all the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ hold (B' is a conjunction of clauses)
- for any assignment to x_1 and x_2 this requires l_1 holds
  i.e. all clauses hold if and only if the clause C hold

SAT ∝ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT
For each clause C of B we construct a number of clauses of B'

- if $C=l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \lor x_1 \lor x_2)$, $(l_1 \lor x_1 \lor \neg x_2)$, $(l_1 \lor \neg x_1 \lor x_2)$, $(l_1 \lor \neg x_1 \lor \neg x_2)$ to B'
- if $C=(l_1 \lor l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \lor l_2 \lor y)$ and $(l_1 \lor l_2 \lor \neg y)$ to B'
- B' holds if and only if both the clauses $(l_1 \lor l_2 \lor y)$ and $(l_1 \lor l_2 \lor \neg y)$ hold
- for any assignment to y this requires $(l_1 \lor l_2)$ holds
  i.e. both clauses hold if and only if the clause C holds

SAT ∝ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT
For each clause C of B we construct a number of clauses of B'

- if $C=l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'
- if $C=(l_1 \vee l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \vee l_2 \vee y)$ and $(l_1 \vee l_2 \vee \neg y)$ to B'
- if $C=(l_1 \vee l_2 \vee l_3)$, we add the clause C to B'

SAT ∝ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT
For each clause C of B we construct a number of clauses of B'

- if $C=l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \lor x_1 \lor x_2)$, $(l_1 \lor x_1 \lor \neg x_2)$, $(l_1 \lor \neg x_1 \lor x_2)$, $(l_1 \lor \neg x_1 \lor \neg x_2)$ to B'

- if $C=(l_1 \lor l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \lor l_2 \lor y)$ and $(l_1 \lor l_2 \lor \neg y)$ to B'

- if $C=(l_1 \lor l_2 \lor l_3)$, we add the clause C to B'

- if $C=(l_1 \lor \ldots \lor l_k)$ and $k>3$, we introduce $k-3$ addition variables $z_1,\ldots,z_{k-3}$ and add the clauses $(l_1 \lor l_2 \lor z_1)$, $(\neg z_1 \lor l_3 \lor z_2)$, $(\neg z_2 \lor l_4 \lor z_3)$, …, $(\neg z_{k-4} \lor l_{k-2} \lor z_{k-3})$, $(\neg z_{k-3} \lor l_{k-1} \lor l_k)$ to B'

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT
For each clause C of B we construct a number of clauses of B'

- if $C=l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'
- if $C=(l_1 \vee l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \vee l_2 \vee y)$ and $(l_1 \vee l_2 \vee \neg y)$ to B'
- if $C=(l_1 \vee l_2 \vee l_3)$, we add the clause C to B'
- if $C=(l_1 \vee \ldots \vee l_k)$ and $k>3$, we introduce $k-3$ addition variables $z_1,\ldots,z_{k-3}$ and add the clauses $(l_1 \vee l_2 \vee z_1)$, $(\neg z_1 \vee l_3 \vee z_2)$, $(\neg z_2 \vee l_4 \vee z_3)$, …, $(\neg z_{k-4} \vee l_{k-2} \vee z_{k-3})$, $(\neg z_{k-3} \vee l_{k-1} \vee l_k)$ to B'
- again all clauses hold if and only if C holds

Coping with NP-completeness

What to do if faced with an NP-complete problem?

Maybe only a restricted version is of interest (which maybe in P)

- e.g. `2-SAT`, `2-colouring` are in P

Seek an exponential-time algorithm improving on exhaustive search

- e.g. backtracking (as in the assessed exercise), branch-and-bound
- should extend the set of solvable instances

For an optimisation problem (e.g. calculating min/max value)

- settle for an approximation algorithm that runs in polynomial time
- especially if it gives a provably good result (within some factor of optimal)
- use a heuristic
  - e.g. genetic algorithms, simulated annealing, neural networks

For a decision problem

- settle for a probabilistic algorithm correct answer with high probability