

# COMP90038 Algorithms and Complexity

## NP-Completeness

Lars Kulik

Lecture 22

Semester 1, 2016

# Concrete Complexity

We have been concerned with the analysis of algorithms' running times (best, average, worst cases).

Our approach has been to give a bound for the **asymptotic** behaviour of running time, **as a function of input size**.

For example, the quicksort algorithm is  $O(n^2)$  in the worst case, whereas mergesort is  $O(n \log n)$ .

# Abstract Complexity

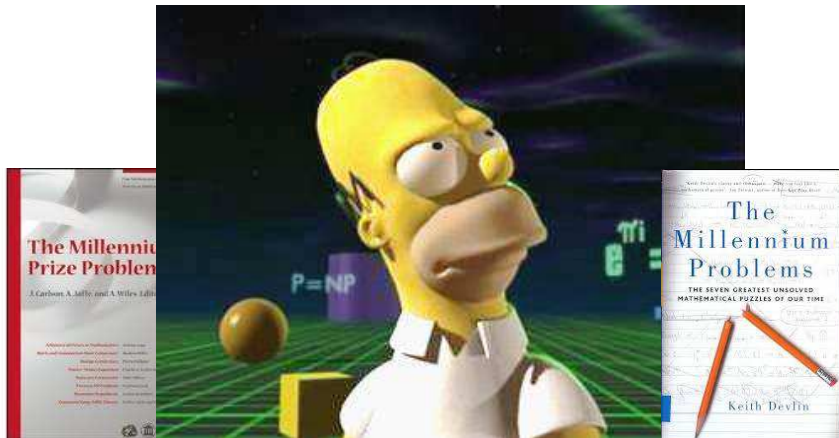
Complexity theory instead asks:

“What is the inherent difficulty of the **problem**?”

How do we know when we have come up with an algorithm which is **optimal** (in the asymptotic sense).

# A Million Dollar Question: Is $P = NP$ ?

This is one of the seven “millennium problems”: The Clay Institute’s seven most important unsolved mathematical problems.



# Who Wants to Be a Millionaire?

The “P versus NP” problem comes from computational complexity theory.

If you could use USD 1,000,000, solve this—and you might well get a bonus Turing Award.

Or solve any of the other millennium problems—all but one remain unsolved.

# Algorithmic Problems

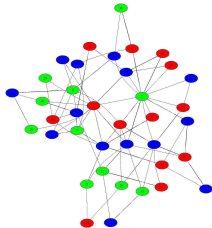
When we talk about a **problem** in computer science we almost always mean a family of **instances** of a general problem.

An **algorithm** for the problem has to work for all possible instances (input).

**Example:** The **sorting** problem—an instance is a sequence of items.

**Example:** The **graph  $k$ -colouring** problem—an instance is a graph.

**Example:** **Equation solving** problems—an instance is a set of, say, linear equations.



# Easy and Hard Problems

A path in a graph  $G$  is **simple** if it visits each node of  $G$  at most once.

Consider this problem for undirected graphs  $G$ :

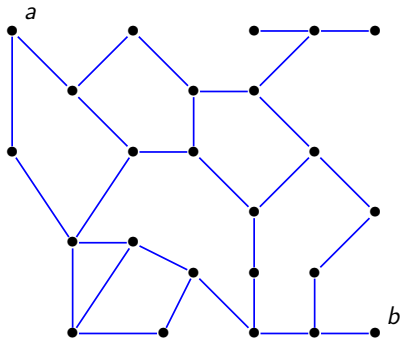
**SPATH**: Given  $G$  and two nodes  $a$  and  $b$  in  $G$ ,  
is there a simple path from  $a$  to  $b$  of length **at most**  $k$ ?

And this problem:

**LPATH**: Given  $G$  and two nodes  $a$  and  $b$  in  $G$ ,  
is there a simple path from  $a$  to  $b$  of length **at least**  $k$ ?

If you had a large graph  $G$ , which of the two problems would you rather have to solve?

# Easy and Hard Problems



There are fast algorithms to solve SPATH.

Nobody knows of a fast algorithm for LPATH.

It is likely that the LPATH problem cannot be solved in polynomial time.

(But we don't know for sure.)

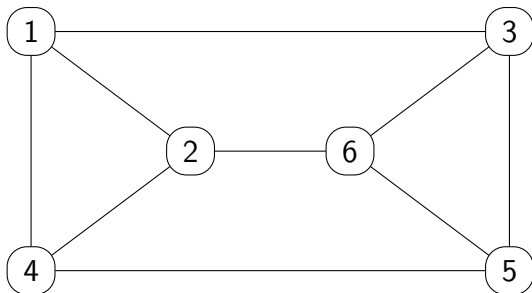


# Hamiltonian Tours

The Hamiltonian tour problem (Lecture 5) again:

**HAM:** In a given graph, is there a path which visits each **node** of the graph once, returning to the origin?

Is there a Hamiltonian tour of this graph?

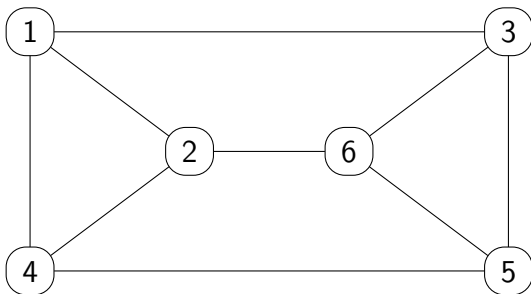


# Eulerian Tours

The Eulerian tour problem is this:

**EUL:** In a given graph, is there a path which visits each **edge** of the graph once, returning to the origin?

Is there a Eulerian tour of this graph?



# More Problems

Try to rank these problems according to inherent difficulty:

- 1 **SUBSET-SUM**: Given a set  $S$  of positive integers and a positive integer  $t$ , is there a subset of  $S$  that adds up to  $t$ ?
- 2 **3COL**: Given a graph  $G$ , is it possible to colour the nodes of  $G$  using only three colours, so that no edge connects two nodes of the same colour?

# Polynomial-Time Verifiability

SUBSET-SUM, 3COL, LPATH, and HAM share an interesting property.

If somebody claims to have a solution (a “yes instance”) then we can quickly test their claim.

In other words, these problems **seem hard to solve**, but solutions allow for **efficient verification**. They are **polynomial-time verifiable**.

That property is shared by a very large number of interesting problems in planning, scheduling, design, information retrieval, networks, games, ...

# Turing Machines

A **Turing machine** has an infinite tape through which it takes its input and performs its computations.

It can

- both read from and write to the tape,
- move either left or right over the tape.

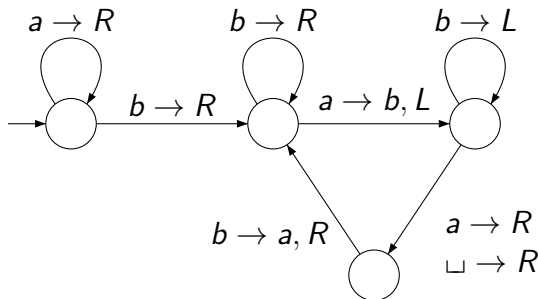
The tape is unbounded in both directions.

A Turing machine may fail to halt.



# Turing Machine Example: Sorting

This (halting) Turing machine sorts a sequence of *as* and *bs*.



The tape alphabet is  $\{\sqcup, a, b\}$ .

# Turing Machines and Computability

Surprisingly, Turing machines appear to have the same “computational power” as any other sensible computing device.

By this we mean that any function that can be implemented in C, Java, Haskell, ... can be implemented on a Turing machine.

The Turing machine has a certain **universality** property: There is a Turing machine which is able to simulate any other Turing machine.

Here we shall assume that Turing machines are used to implement **decision procedures**: algorithms with a yes/no answer.

# Non-Deterministic Turing Machines

One variant of the Turing machine has a powerful **guessing** capability: If different moves are available, the machine will favour a move that ultimately leads to a 'yes' answer.

Adding this kind of **non-determinism** to the capabilities of Turing machines does not change **what** Turing machines can compute.

However, it may have an impact on efficiency.

What a non-deterministic Turing machine can compute in polynomial time corresponds exactly to the class of polynomial-time verifiable problems.

Is there anything it can do in polynomial time that a deterministic machine cannot?



# P and NP

$P$  is the class of problems solvable in polynomial time by a deterministic Turing machine.

$NP$  is the class of problems solvable in polynomial time by a non-deterministic Turing machine.

Clearly  $P \subseteq NP$ . **Is  $P = NP$ ?**

Most computer scientists consider this computer science's

**most important open question**

# Computability Theory

That completes our brief tour of complexity theory.

Sometimes we run into more trouble than mere intractability!

There are practically important computational problems that have  
**no algorithmic solution!**

But that's a topic you'll have to discover in some other subject, on  
**computability theory.**