Computer Systems on the CERN LHC:
How do we measure how well they're doing?

Joshua Dawes
PhD Student
affiliated with CERN & Manchester

My research pages: `http://cern.ch/go/GGh8`

Purpose of this talk

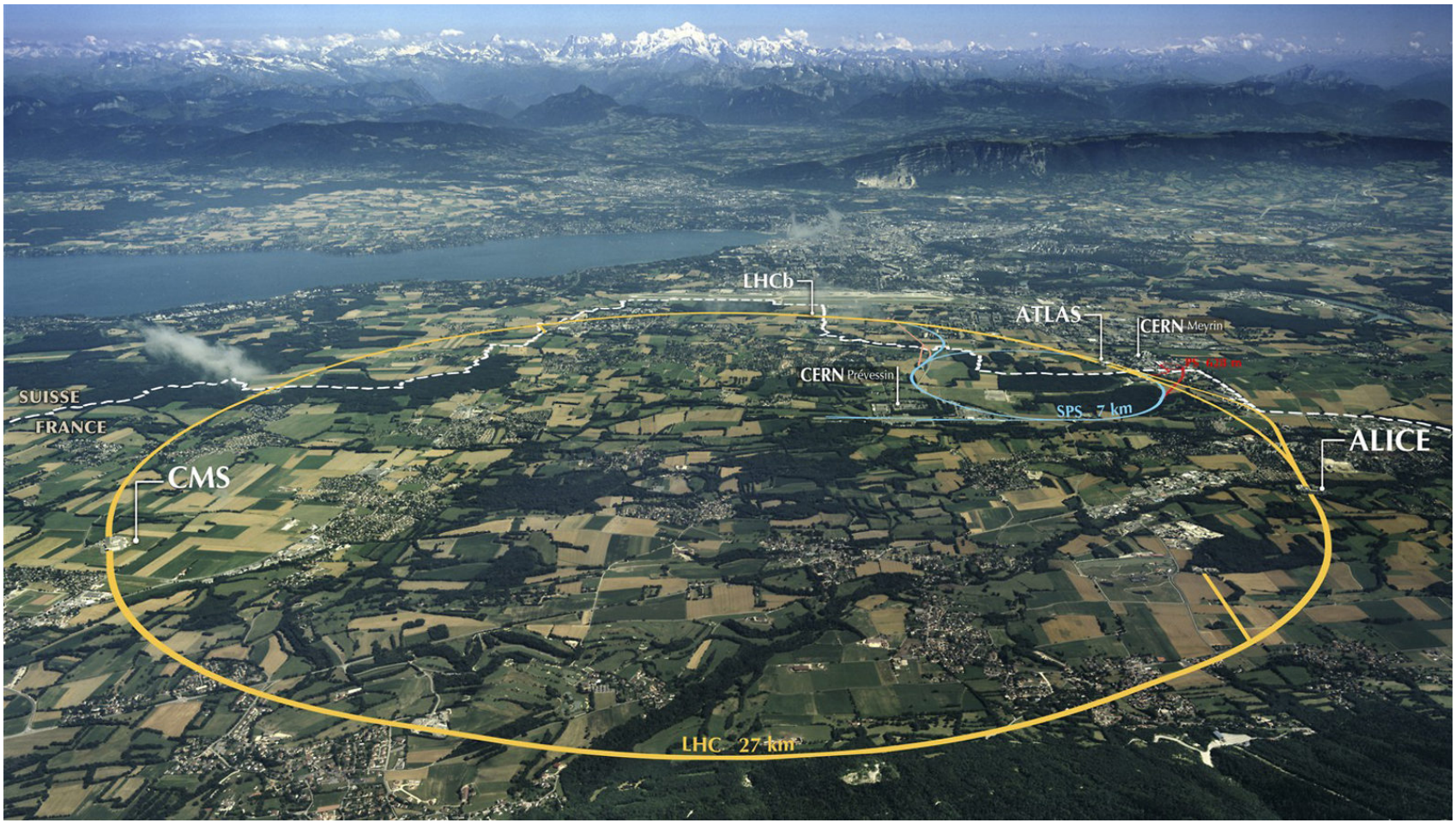
- Principally **for you to learn something!**
- But also **for you to see why the stuff you're learning right now is useful.**
- **Ultimately...** for you to see what Computer Science looks like at the cutting edge :)
- **Not** an exercise in understanding code! That's not what Computer Science is!!

Who am I? What's this talk about?

- I'm a PhD Student on the CMS Experiment at CERN.
- Did A-Levels here, undergrad at Manchester.
- **PhD?** research degree.
- This talk's about a part of my PhD.

The LHC and CMS

- **CERN** - high energy and particle physics research lab in Geneva, Switzerland.
- **LHC** - Large Hadron Collider - 27km circular collider, collides proton beams at 13 TeV, do 11,000 laps per second.
- **CMS** - Compact Muon Solenoid - one of the *detectors* on the LHC ring.
- Observes in the order of 30 million collisions per second.
- It's about 5 stories tall and weighs twice as much as the Eiffel tower...



Credit: Maximilien Brice, CERN

What do we get out of CMS?

- **Events** - collisions! As readings from parts of the detector such as calorimeters.
- **Non-event structure** - includes the alignment and calibrations of the detector.
  What I worked on when I was at CERN as an undergrad.
- We perform **offline reconstruction** by combining **events** and **non-events**.

Topic of my PhD

- **Instrumentation** of CMS computer systems to understand better their behaviour.

State	How the data the program operates with looks.
Control Flow	Which path the program takes. Complex programs can take many...
Verification	Does the program's execution hold some properties? Combination of state and control flow.

Key question - **what is instrumentation?**

Instrumentation

- *Verb:* **Instrument** means to place *instruments* into something that will be a means of measuring some data.
- *Noun:* **Instrumentation** is the process of deciding where to place the instruments, and doing so.

An Analogy

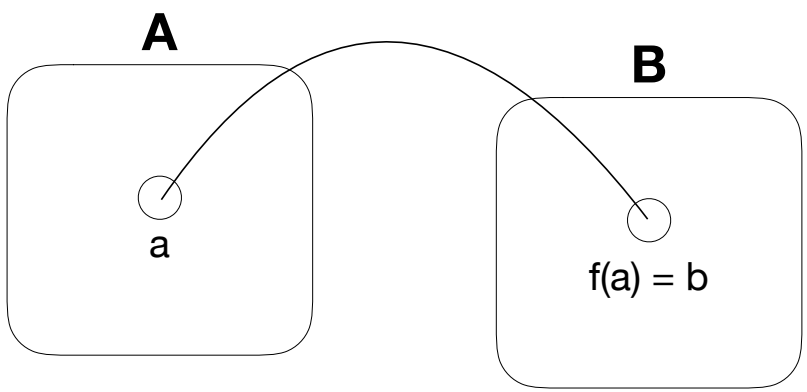
Imagine someone's painting a picture.

- You want to build up an idea of how the painter is doing.
- To do this, you ask the painter.
- Every time you ask, they have to stop to answer you.
  Each time you ask = an instrument
  Deciding when to ask = part of instrumentation
- This *delay* is the *overhead*.

We choose an instrumentation strategy to minimise overhead. Hence, we have an optimisation problem.

Optimisation - Preliminaries

- **Function**: Rule mapping things from one set of things to another.
- $a \in A$ means a is a *member* of A.
- We write $f : A \to B$ for a function sending elements $a \in A$ to $f(a) = b \in B$.
- Functions you're used to are $\mathbb{R} \to \mathbb{R}$.
- We can do calculus with real-valued functions...
- Why is this useful in optimisation?



Optimisation ≡ finding stationary points (sometimes)

- Set $f'(x_0) = 0$, solve for x_0.
- Use the second-derivative tests you've learned to find which x_0 yields maximal $f(x_0)$.
- Gives a *local* stationary point... why not *global*?
- Quite often the thing we optimise isn't a straightforward function.

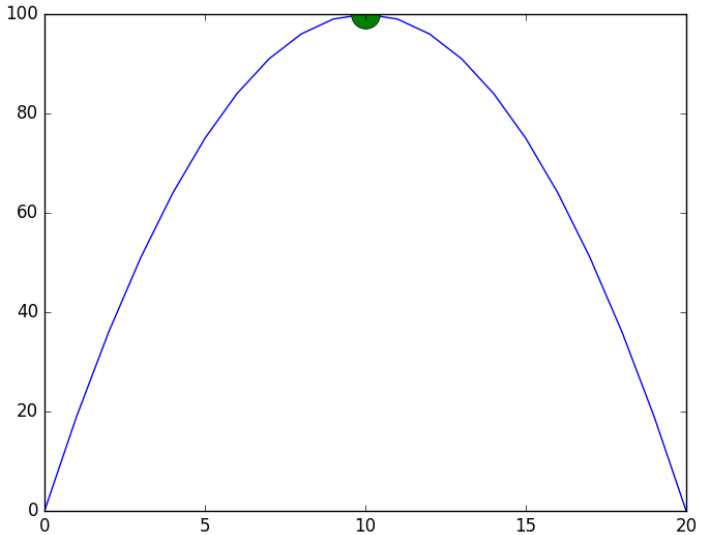


Figure: $f(x) = 20x - x^2$

We need some theory!

- The theory of Computation
- NP-completeness
- Optimisation $\equiv$ Search

Then, with all that in hand, back to **Instrumentation as Optimisation**.

Computation

- *Verb:* **Compute**... pretty self-explanatory.
- *Noun:* **Computation** is the process happening when one computes.

We need to reason about computation formally.

"Models"

- In the maths you've studied:
  - *Dynamical systems* can be modelled by *differential equations*.
  - There are then rules you can apply to reason about the dynamical system.
  - *Phase portrait analysis*, *closed-form solutions*...
  - The differential equation helps form a *model*.
- We need the same in computation.
- So we have *models of computation*.

A Model: Turing Machines for Decision Problems

- **Roughly:** *Abstract* computers, with infinite memory.
- We'll miss out the details.
- This is the foundation we need for the rest of the talk.
- We consider decision problems. The input is essentially a question. The output is **yes**, **no** or **don't know**.

Optimisation problems can be expressed as decision problems.

Bounds on Computation

- Turing machines take an input x of size $|x| = n \in \mathbb{N}$.
- They compute a function on input x in time **bounded by** $g(n)$.
- Just means if the input has size $|x| = n$, then the Turing Machine will compute the answer in a number of steps $\leq g(n)$.
- The bound $g(n)$ is really important.

We divide problems into classes based on the types of functions we can use to bound their decision computation.

Examples

- We take a **polynomial** of degree k to be a function $f : \mathbb{N} \to \mathbb{R}$ with

$$f(n) = a_0 + a_1 n^1 + \cdots + a_k n^k$$

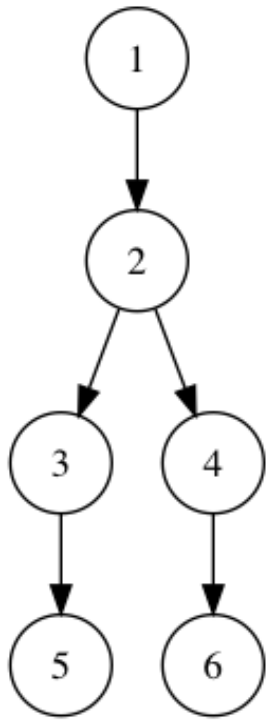
  such that $a_i \in \mathbb{R}$.
- *These don't grow that quickly. Take* $f(n) = n^2$*; then* $f(10) = 100$ *and* $f(100) = 10,000$.

Examples

- And we take an **exponential** with base 2 to be a function $g : \mathbb{N} \to \mathbb{R}$ with
$$g(n) = 2^n.$$
- *These grow very quickly. Take* $g(n) = 2^n$*; then* $f(10) = 1,024$ *and* $f(100) \approx 1 \times 10^{30}$*.*
- If computation is bounded most tightly by an exponential, we are in trouble.

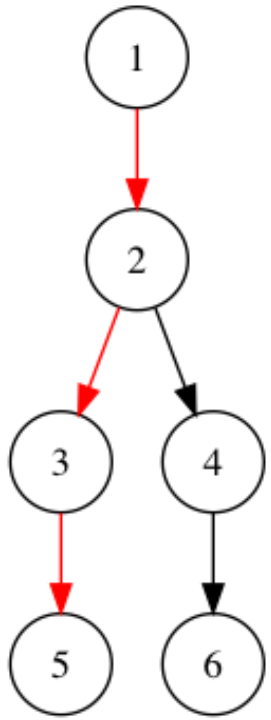
Determinism vs Non-determinism

- Represent the sequence of steps a Turing Machine goes through as a tree.
- Depth of tree $\leq g(n)$.
- The root is the first step taken for every input.
- Leaves are final states of the Turing Machine.
- Rigorous definitions of (Non-)determinism can be given using Automata theory.



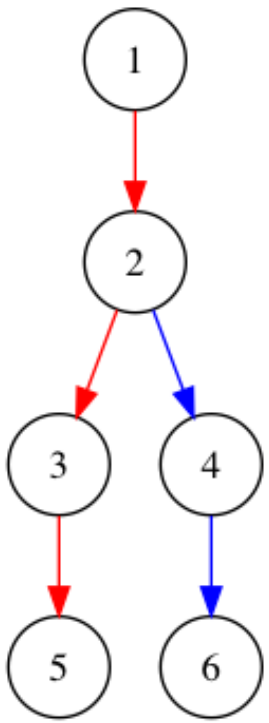
Deterministic Turing Machine

Deterministic: for one input, there is **only one** path of computation.



Non-Deterministic Turing Machine

- **Non-Deterministic**: for one input, there are **multiple** paths of computation.
- We say an instance of a decision problem is positive (ie, the answer is yes) if:
- *There exists a path of computation which outputs yes.*
- **Key**: each run of a Turing Machine explores a single path of computation.



(Non-)Deterministic Complexity Classes

PTIME = problems that can be decided by a deterministic Turing Machine in a number of steps bounded by a polynomial.

NPTIME = same as above, but by a non-deterministic Turing Machine.

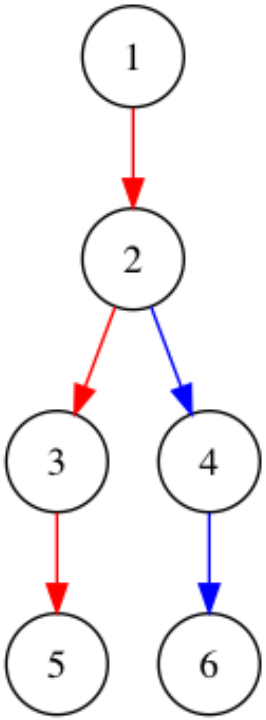
- Problems in NPTIME take much longer to solve. Why?
- We care about solving a problem deterministically. Again, why?
- How do we **determinise** a process?

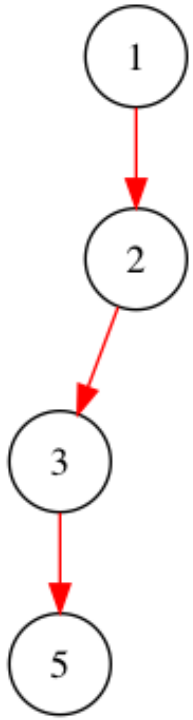
Determinising NPTIME gives EXPTIME

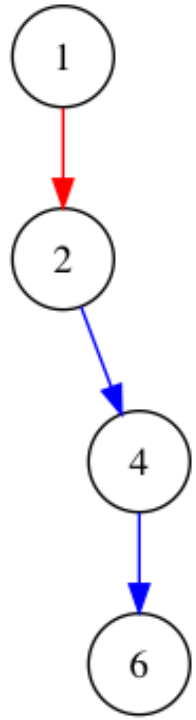
EXPTIME = problems that can be decided by a deterministic Turing Machine in a number of steps bounded by an exponential.

- Performing non-deterministic computation deterministically results in exponential time complexity. Why?

Because...







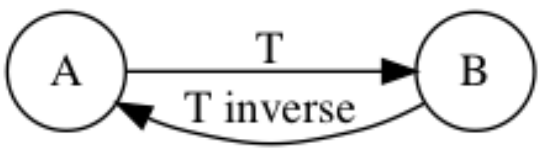
If computation is a binary tree, each final vertex $\equiv$ one path of computation.

There are $\leq 2^{g(n)}$ final vertices.

Deterministic computation must enumerate **all of them**.

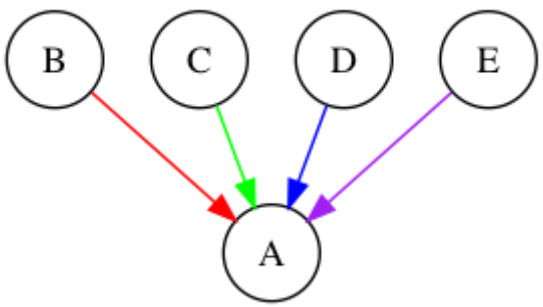
One last piece: Reduction

- High level description...
- **Reduction** = construct a transformation T from problem A to B.
- Solve B.
- Apply T^{-1}.
- This gives a solution to A.
- T is a **reduction** of A to B.
- **Key:** the decision problem A yields *yes* **if and only if** B does.



All of this leads to: NP-Completeness

- Let A be a decision problem.
- If, for every problem B in **NPTIME**, there is a T such that T reduces B to A, A is **NP-hard**.
- **NP-Complete = NP-hard + in NP**
- If a problem is NP-Complete, **that is not a good thing.**



Significance?

- We don't know whether
  1. We're not good enough at Computer Science, or
  2. It's impossible to solve NP-Complete problems in PTIME...
- It's a very big research question.
- The famous **P = NP?**
- We can use **Heuristics** to reduce the Search Space - *little bits of knowledge specific to the problem*.
- Search space can be reduced to the extent of moving the problem to PTIME.

Instrumentation by Optimisation

- We have an optimisation problem:

> Decide when to inspect the program's state such that intrusion is minimal *and* we can reconstruct as much of the runtime as we need.

- Which properties of the runtime do we want to reconstruct?
- How can we ask questions about the properties?
- One (recent) approach is to *admit defeat* and let some data go. We then compute probabilities of what we think the data we missed could be.

In Context: Instrumenting Physics systems at CERN

- On the CMS Experiment, we have many large systems.
- They work with Event (Physics) and Non-event (Alignment/Calibrations) data.
- When data comes out of the detector, they get it ready for injection into **reconstruction**.

Distributed, Reactive Systems

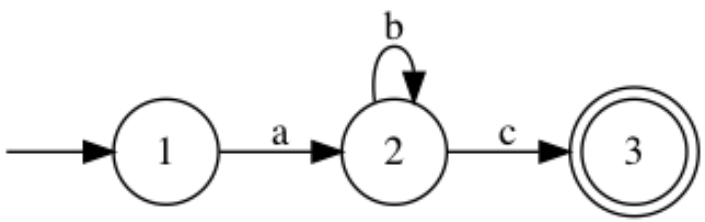
- They are often web services, so **distributed** (multiple computers, working together) and **reactive** (react to their environment).
- Reconstructing behaviour of reactive, distributed systems is very hard.
  One of the main topics of my research, actually, but out of the scope of this talk.

NP-Completeness in this Context

- The instrumentation optimisation problem could be NP-Complete.
- Size of the optimisation/decision problem $\propto$ size of the system.
- **Why? Because the search space of the optimisation problem is generated by the system being instrumented.**

Reasoning about Optimality?

- A model of the combination of a system's behaviour and the measurements taken after instrumentation is needed.
- In general, there's heavy use of automata and graph theory.
- **More sophisticated automata help prove properties of instrumentation methods...**
- Out of the scope of this talk.



What we've learned

- A bit about CERN.
- Some things about optimisation
- ...and why it's so hard.
- Reconstructing behaviour, as an optimisation problem.
- Reasoning about optimality.

Thank you for listening! :)