# CoCoNuT - Complexity - Lecture 1

N.P. Smart

Dept of Computer Science
University of Bristol,
Merchant Venturers Building

March 19, 2015

# Outline

Is PATH in P?

Problems With Easily Checkable Solutions

The Class NP

Reductions, and NP-Completeness

# Why are TM's So Important

Almost any model of computing we come up with (especially practical ones) are equivalent to a TM.

### Church-Turing Thesis

Anything that can be computed, can be computed by a Turing Machine

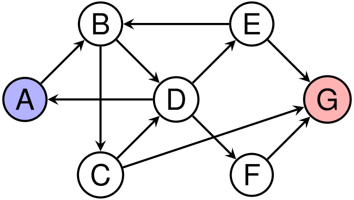
More importantly for almost all models of computation the notion of P is the same as P on a Turing Machine.

So P really is quite fundamental as a class of problems in computing.

# PATH $\in$ P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$

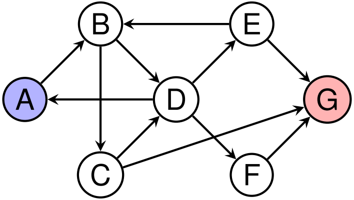



- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.
- A naïve algorithm would be to try every possible path from $s$ to $t$ in turn to see if that path exists.
- But if $G$ has $m$ vertices, in the worst case this could involve checking $\sim m^m$ paths!!!!

# PATH $\in$ P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$

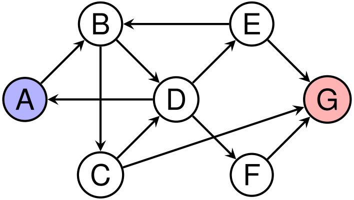



- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.
- A naïve algorithm would be to try every possible path from $s$ to $t$ in turn to see if that path exists.
- But if $G$ has $m$ vertices, in the worst case this could involve checking $\sim m^m$ paths!!!!

# PATH $\in$ P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$

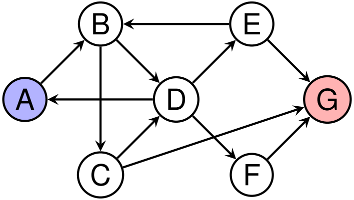



- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.
- A naïve algorithm would be to try every possible path from $s$ to $t$ in turn to see if that path exists.
- But if $G$ has $m$ vertices, in the worst case this could involve checking $\sim m^m$ paths!!!!

# PATH $\in$ P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$




- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.
- A naïve algorithm would be to try every possible path from $s$ to $t$ in turn to see if that path exists.
- But if $G$ has $m$ vertices, in the worst case this could involve checking $\sim m^m$ paths!!!!

# PATH $\in$ P

## A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex $s$.
2. Repeat until no further vertices are found:
   - Check all the edges in $G$. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If $t$ is marked, accept; otherwise reject.

Assume the input graph $G$ is on $m$ vertices and is provided as an adjacency matrix, so the input size $n = O(m^2)$.

# PATH $\in$ P

## A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex $s$.
2. Repeat until no further vertices are found:
   - Check all the edges in $G$. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If $t$ is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Step 2 is repeated at most $m$ times and checks at most $m^2$ edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed $k$ (depending on the computational model).
- So the running time of the algorithm is $O(m^{k+3})$, which is poly($n$).

# PATH $\in$ P

## A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex $s$.
2. Repeat until no further vertices are found:
   - Check all the edges in $G$. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If $t$ is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Step 2 is repeated at most $m$ times and checks at most $m^2$ edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed $k$ (depending on the computational model).
- So the running time of the algorithm is $O(m^{k+3})$, which is poly($n$).

# PATH $\in$ P

## A polynomial-time algorithm deciding PATH

On input $\langle G, s, t\rangle$:

1. Place a mark on vertex $s$.
2. Repeat until no further vertices are found:
   - Check all the edges in $G$. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If $t$ is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Step 2 is repeated at most $m$ times and checks at most $m^2$ edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed $k$ (depending on the computational model).
- So the running time of the algorithm is $O(m^{k+3})$, which is poly$(n)$.

# PATH $\in$ P

## A polynomial-time algorithm deciding PATH

On input $\langle G, s, t\rangle$:

1. Place a mark on vertex $s$.
2. Repeat until no further vertices are found:
   - Check all the edges in $G$. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If $t$ is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Step 2 is repeated at most $m$ times and checks at most $m^2$ edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed $k$ (depending on the computational model).
- So the running time of the algorithm is $O(m^{k+3})$, which is poly($n$).

# Other examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  PRIMES $= \{x \in \{0,1\}^* \mid x$ is a prime number written in binary$\}$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

- Every context-free language is in P.

- Other examples include evaluation of circuits, finding shortest paths, pattern matching, linear programming, ...

# Other examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  $$\text{PRIMES} = \{x \in \{0,1\}^* \mid x \text{ is a prime number written in binary}\}$$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

- Every context-free language is in P.

- Other examples include evaluation of circuits, finding shortest paths, pattern matching, linear programming, ...

# Other examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  $$\text{PRIMES} = \{x \in \{0,1\}^* \mid x \text{ is a prime number written in binary}\}$$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

- Every context-free language is in P.

- Other examples include evaluation of circuits, finding shortest paths, pattern matching, linear programming, ...

# Problems With Easily Checkable Solutions

We can think of P as the set of problems for which it is easy to come up with a solution.

We will now consider problems for which once you have a solution it can be checked easily.

The former problmes are the ones which students like in exams (easy to do).

The latter are the ones which lecturers like (easy to mark)

We can think of a solution as a proof of some statement.

Thus we are essentially asking: Are some proofs (which can be easily verified) hard to find?

- i.e. Is Maths hard?

# Sudoku

Sudoku is a problem which is hard to solve, but easy to check.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

Pics: Wikipedia/Sudoku

# FACTORING

FACTORING

$$= \{\langle x, y\rangle \mid x \text{ is an integer with a prime factor lower than } y\}.$$

For example, $15, 4$ is in the language, but $15, 2$ is not.
For any $x$, a prime factor $z$ of $x$ less than $y$ can be used to prove whether $\langle x, y\rangle$ is in the language.

- We can check primality of $z$ in poly-time via the AKS algorithm.

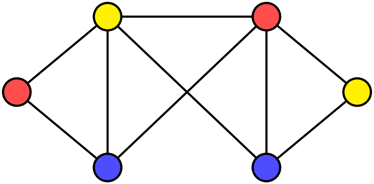
$\overline{\text{FACTORING}}$

$$= \{\langle x, y\rangle \mid x \text{ is an integer with no prime factor lower than } y\}.$$

Given some claimed prime factors $p_1, \ldots, p_t$ of $x$, we can multiply them together to determine whether we get $x$.
If so, we check whether the smallest of them is larger than $y$.
The value $t$ is bounded by $\log x$, so the proof is poly-size in the input.

# Graph Colouring

We say a graph can be properly $k$-coloured if each vertex can be assigned one of $k$ colours, such that all pairs of adjacent vertices have different colours.



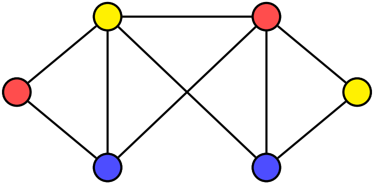
We can formalise this decision problem as the language

$$\begin{aligned} &3\!-\!\text{COLOURING} \\ &\quad = \{\langle G\rangle \mid G \text{ is a graph which can be properly 3-coloured}\}. \end{aligned}$$

Given a graph, we can be convinced that it can be properly 3-coloured by being given a 3-colouring and checking that it is proper; so a solution to 3-COLOURING can be checked quickly.

# Graph Colouring

We say a graph can be properly $k$-coloured if each vertex can be assigned one of $k$ colours, such that all pairs of adjacent vertices have different colours.

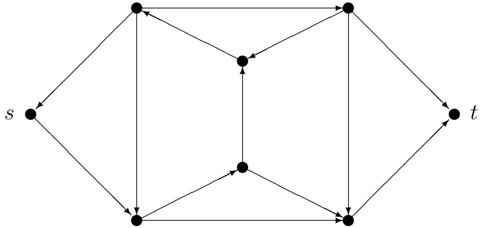


We can formalise this decision problem as the language

$$\begin{aligned} & 3\text{–COLOURING} \\ & \quad = \{\langle G \rangle \mid G \text{ is a graph which can be properly 3-coloured}\}. \end{aligned}$$

Given a graph, we can be convinced that it can be properly 3-coloured by being given a 3-colouring and checking that it is proper; so a solution to 3-COLOURING can be checked quickly.

# Hamiltonian Path




$$HAMPATH = \{(G, s, t) \mid G \text{ isdirected graph with a Hamiltonian path from } s \text{ to } t\}$$

A Hamiltonian path is one which visits each vertex exactly once

Easy to check that $(G, s, t) \in HAMPATH$ when given a path;

# Efficiently Verify Solutions

Many computational problems:

- Can be solved by brute-force, testing exp. many *candidates*.
- Verification of desired property on a candidate is easy.

We can formalise problems which are easy to verify a solution in the following way:

**Definition: Verifier**

A verifier for a language $\mathcal{L}$ is a Turing machine $V$ such that

$$\mathcal{L} = \{x \mid \text{there exists a string } c \text{ such that } V \text{ accepts } \langle x, c\rangle\}.$$

# Efficiently Verify Solutions

Many computational problems:

- Can be solved by brute-force, testing exp. many *candidates*.
- Verification of desired property on a candidate is easy.

We can formalise problems which are easy to verify a solution in the following way:

### Definition: Verifier

A verifier for a language $\mathcal{L}$ is a Turing machine $V$ such that

$$\mathcal{L} = \{x \mid \text{there exists a string } c \text{ such that } V \text{ accepts } \langle x, c \rangle\}.$$

# Efficiently Verify Solutions

We think of $c$ as a proof or witness or certificate that $x \in \mathcal{L}$, which $V$ can check.

$V$ should accept a correct proof (correctness), but not be fooled by any claimed incorrect proof (soundness).

A polynomial-time verifier is a verifier which runs in time polynomial in the length of the input $x$, i.e. $O(|x|^k)$ for some fixed $k$.

**Definition: NP**

NP is the class of languages which have a polynomial-time verifier.

# Efficiently Verify Solutions

We think of $c$ as a proof or witness or certificate that $x \in \mathcal{L}$, which $V$ can check.

$V$ should accept a correct proof (correctness), but not be fooled by any claimed incorrect proof (soundness).

A polynomial-time verifier is a verifier which runs in time polynomial in the length of the input $x$, i.e. $O(|x|^k)$ for some fixed $k$.

**Definition: NP**

NP is the class of languages which have a polynomial-time verifier.

# More on NP

P $=$ class of languages that can be **decided** "quickly"

NP $=$ class of languages that can be **verified** "quickly"

## Definition: co-C

For a class of languages **C**, we define **co-C** as the class of all complements $\overline{A}$ of languages $A$ in C.

See our examples FACTORING and $\overline{\text{FACTORING}}$ from earlier.

### Properties

- P $=$ co-P
- NP $\stackrel{?}{=}$ co-NP

# NP Examples

$HAMPATH \in \text{NP}$

$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\} \in \text{NP}$

$PRIMES \in \text{co-NP}$

- Actually: $PRIMES \in \text{NP}$ (not obvious)
- $PRIMES \in \text{P}$ (shown in 2003)

$\overline{HAMPATH} \overset{?}{\in} NP$

$SAT \in \text{NP}, \quad SAT \overset{?}{\in} \text{co-NP}$

# P vs. NP

Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

Also, every language in NP is also in EXP. If there exists an $m$-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.

So P $\subseteq$ NP $\subseteq$ EXP.

It is not known whether P $=$ NP and this question is considered the most important open problem in computer science!

- Resolving it would win you everlasting fame (as well as $1M).

# P vs. NP

Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

Also, every language in NP is also in EXP. If there exists an $m$-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.

So P $\subseteq$ NP $\subseteq$ EXP.

It is not known whether P $=$ NP and this question is considered the most important open problem in computer science!

- Resolving it would win you everlasting fame (as well as $1M).

# P vs. NP

Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

Also, every language in NP is also in EXP. If there exists an $m$-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.

So P $\subseteq$ NP $\subseteq$ EXP.

It is not known whether P $=$ NP and this question is considered the most important open problem in computer science!

- Resolving it would win you everlasting fame (as well as $1M).

# P vs. NP

Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

Also, every language in NP is also in EXP. If there exists an $m$-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.

So P $\subseteq$ NP $\subseteq$ EXP.

It is not known whether P $=$ NP and this question is considered the most important open problem in computer science!

- Resolving it would win you everlasting fame (as well as $1M).

# $P \stackrel{?}{=} NP$?

$$P \stackrel{?}{=} NP$$

*"Can every problem whose solution is quickly verifiable be solved quickly?"*

Implications?

# Reducibility

Informally: If $A$ *reduces* to $B$ then $B$ is "harder" than $A$ (cf. undecidability)

## Definition : Poly-Time Computable

$f : \Sigma^* \to \Sigma^*$ is **polynomial-time computable** if there is a poly-time TM, which on input $w$ halts with $f(w)$ on its tape.

## Definition : Poly-Time Reducible

A language $A$ is **polynomial-time reducible** to $B$ if there is a poly-time computable $f : \Sigma^* \to \Sigma^*$ with

$$w \in A \quad \text{iff} \quad f(w) \in B.$$

We write $A \leq_p B$.

Cannot necessarily "find" $f$, just says one exists!

# Reducibility

Informally: If $A$ *reduces* to $B$ then $B$ is "harder" than $A$ (cf. undecidability)

## Definition : Poly-Time Computable

$f : \Sigma^* \to \Sigma^*$ is **polynomial-time computable** if there is a poly-time TM, which on input $w$ halts with $f(w)$ on its tape.

## Definition : Poly-Time Reducible

A language $A$ is **polynomial-time reducible** to $B$ if there is a poly-time computable $f : \Sigma^* \to \Sigma^*$ with

$$w \in A \quad \text{iff} \quad f(w) \in B.$$

We write $A \leq_p B$.

Cannot necessarily "find" $f$, just says one exists!

# Reducibility

Informally: If $A$ *reduces* to $B$ then $B$ is "harder" than $A$ (cf. undecidability)

## Definition : Poly-Time Computable

$f\colon \Sigma^* \to \Sigma^*$ is **polynomial-time computable** if there is a poly-time TM, which on input $w$ halts with $f(w)$ on its tape.

## Definition : Poly-Time Reducible

A language $A$ is **polynomial-time reducible** to $B$ if there is a poly-time computable $f\colon \Sigma^* \to \Sigma^*$ with

$$w \in A \quad \text{iff} \quad f(w) \in B.$$

We write $A \leq_p B$.

Cannot necessarily "find" $f$, just says one exists!

# NP-completeness

Read $A \leq_p B$ as "A is no harder than B" (up to poly-time factors).

- For "someone" but maybe not "me", as depends on knowing $f$.

So

- If $B$ can be solved quickly so can $A$.
- If $A$ cannot be solved quickly neither can $B$.

**Theorem.** If $A \leq_p B$ and $B \in$ P then $A \in$ P.

**Definition: NP-complete**

A language $B$ is **NP-complete** if

- $B \in$ NP, and
- Every $A$ in NP is polynomial-time reducible to $B$

# NP-completeness

Read $A \leq_p B$ as "A is no harder than B" (up to poly-time factors).

- For "someone" but maybe not "me", as depends on knowing $f$.

So

- If $B$ can be solved quickly so can $A$.
- If $A$ cannot be solved quickly neither can $B$.

Theorem. If $A \leq_p B$ and $B \in \mathrm{P}$ then $A \in \mathrm{P}$.

**Definition: NP-complete**

A language $B$ is **NP-complete** if

- $B \in \mathrm{NP}$, and
- Every $A$ in NP is polynomial-time reducible to $B$

# NP-completeness

Read $A \leq_p B$ as "A is no harder than B" (up to poly-time factors).

- For "someone" but maybe not "me", as depends on knowing $f$.

So

- If $B$ can be solved quickly so can $A$.
- If $A$ cannot be solved quickly neither can $B$.

Theorem. If $A \leq_p B$ and $B \in \mathrm{P}$ then $A \in \mathrm{P}$.

### Definition: NP-complete

A language $B$ is NP-**complete** if

- $B \in \mathrm{NP}$, and
- Every $A$ in NP is polynomial-time reducible to $B$

# NP-Hardness

## Definition: NP-hard

A language $B$ is **NP-hard** if

- Every $A$ in NP is polynomial-time reducible to $B$

So $B$ is NP-complete if

- $\forall A \in \text{NP}$ we have $A \leq_p B$ and $B \in \text{NP}$.

and $B$ is NP-hard if

- $\forall A \in \text{NP}$ we have $A \leq_p B$, and we do not know if $B \in \text{NP}$.

# NP-Hardness

## Definition: NP-hard

A language $B$ is **NP-hard** if

- Every $A$ in NP is polynomial-time reducible to $B$

So $B$ is NP-complete if

- $\forall A \in \text{NP}$ we have $A \leq_p B$ and $B \in \text{NP}$.

and $B$ is NP-hard if

- $\forall A \in \text{NP}$ we have $A \leq_p B$, and we do not know if $B \in \text{NP}$.

# NP-completeness

*NP-complete problems are the "hardest" problems in NP*

Theorem. If $B$ is NP-complete and $B \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.

Theorem. If $B$ is NP-complete and $B \leq_p C$, for $C \in \mathrm{NP}$, then $C$ is NP-complete.

Theorem. *(Cook-Levin) SAT* is NP-complete.

Alternative wording:

$$SAT \in \mathrm{P} \quad \text{iff} \quad \mathrm{P} = \mathrm{NP}$$

# NP-completeness

*NP-complete problems are the "hardest" problems in NP*

Theorem. If $B$ is NP-complete and $B \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.

Theorem. If $B$ is NP-complete and $B \leq_p C$, for $C \in \mathrm{NP}$, then $C$ is NP-complete.

Theorem. *(Cook-Levin) SAT* is NP-complete.

Alternative wording:

$$SAT \in \mathrm{P} \text{ iff } \mathrm{P} = \mathrm{NP}$$

# NP-completeness

*NP-complete problems are the "hardest" problems in NP*

Theorem. If $B$ is NP-complete and $B \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.

Theorem. If $B$ is NP-complete and $B \leq_p C$, for $C \in \mathrm{NP}$, then $C$ is NP-complete.

Theorem. *(Cook-Levin) SAT* is NP-complete.

Alternative wording:

$$SAT \in \mathrm{P} \ \text{ iff } \ \mathrm{P} = \mathrm{NP}$$

# NP-completeness

*NP-complete problems are the "hardest" problems in NP*

Theorem. If $B$ is NP-complete and $B \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.

Theorem. If $B$ is NP-complete and $B \leq_p C$, for $C \in \mathrm{NP}$, then $C$ is NP-complete.

Theorem. *(Cook-Levin) SAT* is NP-complete.

Alternative wording:

$$SAT \in \mathrm{P} \text{ iff } \mathrm{P} = \mathrm{NP}$$