Optimisation and Operations Research
Lecture 13: Complexity and the P vs NP problem

Matthew Roughan
<matthew.roughan@adelaide.edu.au>

School of Mathematical Sciences,
University of Adelaide

August 13, 2019
Section 1

Turing machines
The Problem

- Earlier analysis essentially ignored the underlying computer
- Computation time depends a great deal on the computer
  - e.g., can it parallelise some operations?
- So we need a “universal” model computer to create formal arguments about what is possible
- Turing created one, way back when we were just inventing computing
Turing Machines

- An abstract model of a computer
- Turns out that all sufficiently complex computing systems are equivalent in the sense that they can compute the same family of functions:
  - computable functions intuitively have a finite program, that completes in a finite number of steps to the result
  - almost all functions we deal with in math are computable (though maybe not efficiently)
  - there are a few that aren’t
- Turing machines have a few variants, but simplest has:
  - a tape
  - a finite state machine that can write/read from the tape
Simple Turing Machine

- a tape
  - a tape is an idealisation of computer memory
  - imagine a strip of paper on which we can write or erase some symbols (often binary 1s and 0s)
  - the tape can be moved back and forth so that the machine can write and read any point on the tape
- a finite state machine that can write/read from each tape
  - $n$ states, plus “halt”
  - transition function has inputs of current state and current tape value
  - transition causes three outputs:
    - can write over the current bit of the tape
    - it can move the tape
    - the state machine's state can change
- running the machine means setting a set of tape values, and a starting state, and then allowing transitions until “halt” is reached
Our Turing Machine

- Ours will be just a little different (but equivalent)

- Its helpful to separate inputs and outputs from working memory
  - input tape (with the input $p$ – the program – on it)
  - output tape (which we will write the output $x$ on)
  - a working tape
  - a finite state machine that can write/read from each tape

- We’ll call this a universal computer
  - measure complexity by the number of state changes
Section 2

Complexity Nomenclature
Algorithm v Problem complexity

Remember that

- we start with *instances* of a class of problem of given size $n$
- in general think about solving using a Turing Machine
- we can measure the time of an instance, and often calculate the time of the worst case, when a particular *algorithm* is used to solve it
- we often attribute a *problem* with the complexity of the best algorithm, on the worst *instance*
- describe complexity with big-O notation, *e.g.*, $O(n^2)$
General problem descriptions

Common types of (general) problem

**decision** : does a solution exist?

**search** : find a solution.

**counting** : how many solutions exist?

**optimisation** : find the best solution.

The distinction is arbitrary: e.g., we can solve a decision problem by searching for a solution, but it's helpful in thinking about complexity.
Example

**decision**: is $n$ prime?

**search**: factorise $n$ (if it is possible).

**counting**: how many possible factors does $n$ have?

**optimisation**: find the factorisation which has the largest sum of factors.
Example

Given a set of numbers, e.g., \( S = \{-7, -3, -2, 5, 8\} \)

- **decision**: does some subset of \( S \) add to give zero? \textbf{Yes}.
- **search**: find a subset that adds to give zero: \( \{-3, -2, 5\} \)
- **counting**: how many subsets of \( S \) add to give zero? \textbf{1}?
- **optimisation**: find the subset that adds to zero with the least members. \( \{-3, -2, 5\} \)
Example

TSP (Travelling Sale-person’s Problem) variants:

- **decision**: is there a path visiting each city with distance less than $k$?
- **search**: find a path visiting each city with distance less than $k$.
- **counting**: how many paths have distance less than $k$?
- **optimisation**: find the shortest path visiting all cities.
General problem descriptions: example 4 (LP)

Example

Linear programming problems:

- **decision**: is $Ax \leq b$ feasible?  
  *Simplex Phase I, or is there something easier?*

- **search**: find a feasible solution to $Ax \leq b$. *Simplex Phase I*

- **counting**: how many vertices does the region defined by $Ax \leq b$ have?  
  *This could be hard?*

- **optimisation**: maximise $c^T x$ over $Ax \leq b$. *Simplex*
Decision problems

Definition (Decision problem)

A *decision problem* is a problem whose answer is YES or NO.

- Can be viewed as dividing problem instances in member and non-member instances
- Avoids issues of the output size of the problem
- The alternatives above could be described as *function problems* where a more complex result is the output. It seems a richer class of problem, but can always be recast as decision problems, e.g.,

\[ a \times b \]

- can be recast as a set of problems “is \( a \times b = c \)”

- NB: often, we solve decision problems by searching for a solution! Think of the solution as a *check*. 
Tractability

Problems that can be solved in theory (e.g., given large but finite time), but which in practice take too long for their solutions to be useful, are known as *intractable* problems.

We can’t trivially distinguish the intractable and tractable problems, so we often divide them by their asymptotic performance into:

- **Polynomial** means there is an algorithm which takes time $poly(n)$ for some polynomial $p(n)$ on inputs of length $n$.
  - remember this is the worst case performance
  - write it as $O(n^k)$ for some fixed $k$
  - polynomial time algorithms are often treated as equivalent to tractable

- **Exponential** means it takes time *at least* $2^{poly(n)}$.
  - grows faster than any polynomial
  - exponential algorithms are assumed to be intractable
Section 3

P v NP
Deterministic and Non-Deterministic Turing Machines

- A *deterministic* Turing machine is just what we described earlier:
  - given a particular input, its output is “deterministic”
  - people have built them (almost)
  - standard computers are analogous
- A *non-deterministic* Turing machine: it can have a set of rules that give more than one action for a given situation.
  - in a given state, input a given symbol, perform both $A$ and $B$
  - can think of it as
    - getting to try both possibilities
    - being able to guess the correct branch

Non-deterministic Turing machines don’t exist, but are useful for describing algorithm complexity.
P and NP

**Definition (P)**

P is the set of *decision* problems that are *Polynomial time*, *i.e.*, they can be solved by a deterministic Turing machine in polynomial time.

**Definition (NP)**

NP is the set of *decision* problems that are *Non-deterministic Polynomial time*, *i.e.*, they can be solved by a non-deterministic Turing machine in polynomial time.

- NP does **NOT** mean Non-Polynomial
- It actually includes all polynomial-time decision problems, *i.e.*, $P \subseteq NP$
- We *don't know* if it has anything else in it
  - Is $P = NP$?
  - Win $1,000,000$ if you can answer this
Euler Diagrams

P ≠ NP

P = NP

P = NP
NP = Non-deterministic Polynomial time

Ways to think about NP

- NP problems have an efficient (polynomial time) verifier
  - computing the decision might be hard
  - but checking a YES decision is easy
  - e.g., is $n$ prime?
    - factorization might require checking all possible factors
    - given two factors $p, q$ its easy to check $p \times q = n$

assumes the YES result comes with a “proof certificate” (often a solution) which can be checked.

- They can be solved in polynomial time by a non-deterministic Turing machine using the following approach
  1. Guess a solution
  2. Check it

A non-deterministic Turing machine can make the right guess, so compute time is just the time to check the solution.
NP examples

- All $P$ problems
- Graph isomorphism problem
- Integer factorisation
- SAT
A very general class of decision problems is SAT

Definition (SAT)

A (Boolean) *satisfiability* (SAT) problem has $n$ Boolean variables $x_1, \ldots, x_n$ and a Boolean formula $\phi$ involving the variables. The question is whether there is an assignment (of TRUE and FALSE) to the variables, such that $\phi(x_1, \ldots, x_n) = TRUE$, i.e., we satisfy the formula.

Example

One variable $x_1$ and Boolean formula

$$\phi(x) = x_1 \land \neg x_1$$

where $\land = \text{AND}$ and $\neg = \text{NOT}$, is *not satisfiable* because

$$\text{TRUE AND NOT TRUE} = \text{FALSE}$$
$$\text{FALSE AND NOT FALSE} = \text{FALSE}$$

so there is no value of $x_1$ that leads to $\phi(x_1) = \text{TRUE}$. 
Example

Three variables $x_1$, $x_2$ and $x_3$ and Boolean formula

$$\phi(x) = (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2 \lor x_3) \land \neg x_1$$

where

$$\lor = \text{OR}$$
$$\land = \text{AND}$$
$$\neg = \text{NOT}$$

is satisfied by $x_1 = \text{FALSE}$, $x_2 = \text{FALSE}$, and $x_3$ arbitrarily.
We *think* some problems in NP aren’t in P

- We certainly know some problems in NP for which we have no polynomial-time algorithm *at present*
  - e.g., SAT
  - so we think these might be harder than P
  - a problem is called *NP-hard* if it is at least as hard as the hardest problem in NP
  - we’ll define formally in a moment

- If $P \neq NP$ then NP-hard problems cannot be solved in polynomial time.
We *think* some problems in NP aren’t in P

If $P = NP$, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in “creative leaps,” no fundamental gap between solving a problem and recognizing the solution once it’s found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett. It's possible to put the point in Darwinian terms: if this is the sort of universe we inhabited, why wouldn't we already have evolved to take advantage of it?

*Scott Aaronson*

http://www.scottaaronson.com/blog/?p=122
NP-hard definitions

Definition
A problem $A$ can be reduced to $B$ if we could solve $A$ using the algorithm that solves $B$ as a subroutine.

- If we have a polynomial time reduction (that is one that can be done in polynomial time, excluding the time in the subroutine) then we can efficiently convert one problem into the other.
- So $A$ is no more difficult than $B$.

Definition (NP-hard)
A problem $H$ is NP-hard if every problem $L$ in NP can be reduced in polynomial time to $H$.

- So $H$ is at least as hard as any $L$ in NP.
- Note that an NP-hard problem isn’t necessarily in NP!
Euler Diagrams

NP-hard

P ≠ NP

P = NP

P = NP

complexity

Matthew Roughan (School of Mathematical Sciences, University of Adelaide)

August 13, 2019 27 / 35
NP-complete

Definition (NP-complete)

A problem that is in NP, and in NP-hard is called NP-complete.

- A problem \( p \) in NP is NP-complete if every other problem in NP can be reduced to \( p \) in polynomial time.
- A decision problem is NP-complete if it is in NP, and every problem in NP is reducible to it.
- Cook’s theorem: the Boolean satisfiability problem (SAT) is NP-complete
  - so many proofs of NP-completeness show SAT can be reduced to the problem
Euler Diagrams

NP-hard

NP-complete

NP

P ≠ NP

P = NP

= NP-complete

P = NP

complexity

NP-hard

Matthew Roughan (School of Mathematical Sciences, University of Adelaide)

OORII

August 13, 2019 29 / 35
Example NP-complete problems

- SAT (and many variants)
- *Binary linear programming*
- Set covering
- Hamiltonian circuit
- Graph colouring
- Bin-packing and Knapsack
- TSP problem
- Many others ....

NB: decision versions of the above where its not obvious.
Other important chunks

- If a decision problem is NP-complete, then its optimisation version is NP-hard

- Weirdest case I know
  - The graph isomorphism problem
    - is graph $G_1$ isomorphic to $G_2$
    - is in NP
    - is suspected to be neither in P or NP-complete
    - very recently a *quasi-polynomial* time algorithm was found
call these NPI = NP-intermediate
  - in NP, but not in P or NP-complete

- There are NP-hard problems that are not NP-complete
  - *e.g.*, the halting problem
    - given a program and its input, will it run forever?
    - is undecidable (so not in NP)
    - SAT can be reduced to the halting problem by writing Turing machine program that tries all values
Misconceptions

- NP-complete problems are not the “hardest”
  - they are in NP – some problems aren’t!
    - some problems can’t even be verified in polynomial time
  - decision problems in Presburger arithmetic can take $O(2^{2^n})$, i.e., double exponential time

- Not all instances of NP-complete problems are hard
  - many (even most) instances of some NP-complete problems can be solved in polynomial time
  - complexity refers to worst case

- Problems with an exponential number of possibilities are not all NP-complete
  - counter-example: shortest paths is solvable in $O(n \log n)$ time
Takeaways

- We talked about complexity classes
  - P
  - NP
  - NP-complete
  - NP-hard

  we don’t know if $P = NP$

- At the moment, we can’t solve an NP-complete problem in guaranteed polynomial time
  - integer programming is, in general, NP-complete
    - some of these problems are currently intractable
    - but some restricted subsets of integer programming problems might have polynomial time algorithms
    - others might have good approximations
  - in general, though, we are going to have to be a bit more clever when tackling integer programming problems
    - there is no “one-size-fits-all” like the Simplex for LPs
Something to watch

Watch https: //www.youtube.com/watch?v=YX40hbAHx3s&feature=youtu.be
Further reading I