CoCoNuT: Complexity Notes

N.P. Smart
Adapted from notes by
Richard Jozsa, Ashley Montanaro,
Markus Jalsenius and
Georg Fuchsbauer

March 19, 2015

Contents

Lecture 0

Introduction

These notes are to supplement the slides for the course. Each week you are meant to spend about one sixth of your time on this course. This equates to somewhere between six and severn hours. Two of these hours are lectures, which means four to five hours are reserved for your private study. If you use these notes and lectures as intended you can score high marks in the exam, if not then you risk scoring low marks or even failing.

So how do you use these notes? If you simply master the material in the main part of the notes for each lecture, and attempt to answer the questions in the boxes then you should be able to pass the exam easily. To get high marks you need to *try to* master the material in the *supplementary notes* at the end of some lectures. In addition to the supplementary notes we also sometimes provide some *advanced technical notes*, this is additional material which is not examinable but which provides more background on the material. However, the point of taking this course is not to pass (or get high marks) in the exam; the point is to learn something and expand your mind and think about computer science in new ways. If you manage to do this, then the course is successful[1].

How should you use other material? You are students in a top University. You are meant to be using your time here to capture as much information and understand as much as possible. This is the only time in your life when you are concentrating on learning and expanding your mind and horizons. And you have the facilities in Bristol to do so. Thus you should make use of books in the library, make use of online resources etc. For all courses you should be reading around the subjects studied, finding *your own* linkages between areas in different units, engaging in furter advanced study in specific sub-topics which intrigue you. Basically your heads should always be in a 'book' (although clearly these days such books are more likely to be on a screen).

So how do you use the lectures? You should come to lectures prepared. This means with pen, paper and having read up on what is to be covered *before hand*. This means when we discuss things in lectures you will be able to ask sensible questions and follow the lecture. The lecture is there to set the scene for *your learning outside the lecture theatre*. After the lecture you should read up on the notes you have taken during the lecture and re-read these

[1] If you also get enjoyment out of thinking in these different ways then the course is very successful.

notes provided and attempt any questions. When I was a student I always found re-writing the lecture notes out in my own handwriting helped my understanding, as this slowed my brain down to a point where it started to take in the material.

The object of this first lecture is to set the scene and recap on some material which you have seen in previous units. Thus the notes for this lecture are longer than normal, as it is basically a summary of stuff you should already be familiar with.

0.1 Other Resources

Although this course does not follow a particular textbook closely, the following three books may be particularly helpful.

- *Computational Complexity: A Modern Approach*, Sanjeev Arora and Boaz Barak. Cambridge University Press.

  Encyclopaedic and recent textbook which is a useful reference for almost every topic covered in this course (a first edition, so beware typos!). Also contains much more material than we will be able to cover in this course, for those who are interested in learning more. A draft is available online at `http://www.cs.princeton.edu/theory/complexity/`. This presents a "modern" approach to the subject.

- *Introduction to the Theory of Computation*, Michael Sipser. PWS Publishing Company.

  A more gentle introduction to complexity theory in Part III. In previous years we have used this book extensively, and hence there are lots of copies floating around second hand; and copies in the library. However, it does take a slightly "old fashioned" approach to the subject. It also contains a detailed introduction to automata theory.

- *Computational Complexity: A Conceptual Perspective*, Oded Goldreich, Cambridge University Press.

  This is another book following the "modern" approach. It can get quite technical in places but it has two advantages. Firstly there is extensive discussion in the text about *why* things are important/matter, and secondly the book deals with decision problems and search problems at the same time.

Additional resources you may want to refer to include:

- *Computational Complexity*, Christos Papadimitriou. Addison-Wesley.

  An excellent, mathematically precise and clearly written reference for much of the more classical material in the course (especially complexity classes).

- *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Michael Garey and David Johnson. W. H. Freeman.

  The standard resource for NP-completeness results.

- *Introduction to Algorithms*, Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein. MIT Press and McGraw-Hill.

  A standard introductory textbook to algorithms, known as CLRS.

- There are many excellent sets of lecture notes for courses on computational complexity available on the Internet, including a course at Cambridge[2] and a course at the University of Maryland [3].

These notes have benefited substantially from the above references (the exposition here sometimes follows one or another closely), and also from notes from previous courses on computational complexity by Richard Jozsa, Ashley Montanaro, Markus Jalsenius and Georg Fuchsbauer.

0.2 Motivation

Computational complexity theory is the study of the intrinsic difficulty of computational problems. It is a young field by mathematical standards; while the foundations of the theory were laid by Alan Turing and others in the 1930's, computational complexity in its modern sense has only been an object of serious study since the 1970's, and many of the field's most basic and accessible questions remain open.

In general, we will be concerned with the question of which problems we can solve, given that we have only limited resources (e.g. restrictions on the amount of time or space available to us). Rather than consider individual problems (e.g. "is the Riemann hypothesis true?"), it will turn out to be more fruitful to study families of problems of growing size (e.g. "given a mathematical claim which is n characters long, does it have a proof which is at most n^2 characters long?"), and study how the resources required to solve the problem in question grow with its size. This perspective leads to a remarkably intricate and precise classification of problems by difficulty.

Computational Complexity – What is it?

We know that some problems are uncomputable (eg Turing's halting problem) but a vast variety of problems are computable. They come in many different guises ranging from practical issues:

- e.g. given a map of n cities with distances between them, find the shortest tour that visits all cities;

to recreational issues (games):

[2] `http://www.cl.cam.ac.uk/teaching/1112/Complexity/`
[3] `http://www.cs.umd.edu/~jkatz/complexity/f11/`

- e.g. generalised instant insanity – given n cubes with faces coloured by n colours, can they be stacked in a vertical column so the each colour appears exactly once on each of the four sides of the tower;

to more abstract mathematical issues:

- e.g. given an integer K (with n digits) find a factor of it.

All computational tasks can be recast in a standard mathematical form, which provides a uniform basis for their analysis. This is achieved by specifying a *model of computation*. Papadimitriou writes "Computational problems are not only things that have to be solved, they are also objects that can be worth studying". A basic feature of study is "*Complexity*" – an intuitive but a priori rather vague concept. Part of our task will be to isolate various precise definitions that are useful in certain desired contexts.

Computational tasks (as we'll define them) have an *input size*. This is essentially n in each of the above. Computational complexity theory studies how the computational "effort" to perform the task grows as a function of the input size.
"Computational effort" = usage of "computational resources". For us, computational resources will be time (number of computational steps) and space (amount of computer memory needed to do the task).

We will be interested mainly in broad distinctions between growth rates such as polynomial (poly) growth vs. exponential (exp) growth with input size. This will make our measures of complexity of a task independent of the computational model used or its specific implementation i.e. the complexity will be an intrinsic property of the task itself. Also such broad distinctions lead to a nice abstract theory of complexity – a recent subject which began to be intensively researched only in the 1970's.

Computational Complexity – Why?

Even when a problem is computationally solvable in principle it may not be solvable in practice; it may require inordinate amounts of computational resources (e.g. more time than the age of the universe). The sort of questions which the theory addresses range from the immediately practical to the deeply philosophical, including:

- Assessing how hard a problem is, in terms of resource consumption;
- Are there problems which cannot be solved by computer?
- Are some types of computational resource more powerful than others?
- How can different computational resources can be traded off against each other?
- Does giving computers access to random numbers, or communication, allow more complex problems to be solved efficiently.
- Can we automate the process of discovering mathematical theorems?

- Is there an efficient algorithm for register allocation in a computer's central processing unit?
- Can all algorithms be written to run efficiently on a computer with many processors?
- How quickly can we multiply two $n \times n$ matrices?
- Identification of "hard" computational tasks. These have important applications in modern cryptography such as data authentication and public key cryptosystems.

Complexity theory is an exciting and currently very active area of research. In 2000 the Clay Mathematical Institute (USA) identified 7 most significant open problems, to celebrate mathematics in the new millennium. One of these is the "P vs. NP" problem of complexity theory. They offer a reward of $ 1 million for its solution!

0.3 Notation

We collect here some notation which will be used throughout the course. Σ will usually denote an *alphabet*, i.e. a finite set of symbols. Notable alphabets include the binary alphabet $\{0,1\}$; an element $x \in \{0,1\}$ is known as a bit. A string over Σ is an ordered sequence (possibly empty) of elements of Σ. For strings σ, $|\sigma|$ denotes the length of σ. Σ^k denotes the set of length k strings over Σ, and Σ^* denotes the set of all finite length strings over Σ, i.e. $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$. For $\sigma \in \Sigma$, the notation σ^k denotes the string consisting of k σ's (e.g. $0^3 = 000$).

We will sometimes need to write strings in lexicographic order. This is the ordering where all strings are first ordered by length (shortest first), then strings of length n are ordered such that $a \in \Sigma^n$ comes before $b \in \Sigma^n$ if, for the first index i on which $a_i \neq b_i$, a_i comes before b_i in Σ. For example, lexicographic ordering of the set of nonempty binary strings is $\{0, 1, 00, 01, 10, 11, 000, \dots\}$.

We use the notation $A \subseteq B$ (resp. $A \subset B$) to imply that A is contained (resp. strictly contained) within B. $A \Delta B$ is used for the symmetric difference of sets A and B, i.e. $(A \cup B) \backslash (A \cap B)$. We let $[n]$ denote the set $\{1, \dots, n\}$. For $x \geq 0$, $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and ceiling of x respectively, i.e. $\lfloor x \rfloor = \max\{n \in \mathbb{Z} : n \leq x\}$, $\lceil x \rceil = \min\{n \in \mathbb{Z} : n \geq x\}$.

We will often need to encode elements $s \in S$, for some set S, as binary strings. We write $\underline{s}$ for such an encoding of s, which will usually be done in some straightforward manner. For example, integers $x \in \mathbb{N}$ are represented as $\underline{x} \in \{0,1\}^*$ by just writing the digits of x in binary.

Later on, we will need to compare running times of different Turing machines and/or algorithms. In order to do so while eliding irrelevant details, an important tool will be asymptotic (big-O) notation, which is defined as follows. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that:

- $f(n) = O(g(n))$ if there exist $c > 0$ and integer n_0 such that for all $n \geq n_0$, $f(n) \leq c\, g(n)$, i.e. "the growth rate of f is at most equal to that of g".

- $f(n) = \Omega(g(n))$ if there exist $c > 0$ and integer n_0 such that for all $n \geq n_0$, $f(n) \geq c\,g(n)$. i.e. "the growth rate of f is at least equal to that of g".
- $f(n) = \Theta(g(n))$ if there exist $c_1, c_2 > 0$ and integer n_0 such that for all $n \geq n_0$, $c_1\,g(n) \leq f(n) \leq c_2\,g(n)$. i.e. "the growth rate of f is roughly equal to that of g".
- $f(n) = o(g(n))$ if, for all $\epsilon > 0$, there exists integer n_0 such that for all $n \geq n_0$, $f(n) \leq \epsilon g(n)$. i.e. "the growth rate of f is nothing compared to that of g".
- $f(n) = \omega(g(n))$ if, for all $\epsilon > 0$, there exists integer n_0 such that for all $n \geq n_0$, $g(n) \leq \epsilon f(n)$. i.e. "the growth rate of g is nothing compared to that of f".

Thus: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$; $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; and $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$. The notations $O, \Omega, \Theta, o, \omega$ can be viewed as asymptotic versions of $\leq, \geq, =, <, >$ respectively. Beware that elsewhere in mathematics the notation $f(n) = O(g(n))$ is sometimes used for what we write as $f(n) = \Theta(g(n))$. For example, in our notation $n^2 = O(n^3)$.

Some simple examples:

- **Polynomials:**

  If $f(n)$ is a polynomial of degree k then $f(n) = O(n^k)$ and $f(n) = o(n^{k+1})$.
  Examples: $f(n) = 6n^3 + 2n^2 + 20n + 45$ has degree 3. Show that $f(n) = O(n^3)$, $f(n) = o(n^4)$ and $f(n)$ is not $O(n^2)$.
  For general powers: if $0 < k_1 < k_2$ then $n^{k_1} = o(n^{k_2})$ e.g. $\sqrt{n} = o(n)$.

- **Logarithms:**

  Recall: $y = \log_a x$ means "y is the power of a that gives x" i.e. $a^y = x$. So $a^{\log_a x} = x$. e.g. $\log_2 32 = 5$ because $2^5 = 32$. Also $\lceil \log_2 n \rceil$ is the number of binary digits when n is written in binary.

  We have (for any base a) $\frac{\log n}{n} \to 0$ but even more:

  $$\frac{(\log n)^k}{n^c} \to 0 \quad \text{for all } k > 0 \text{ (however large) and all } c > 0 \text{ (however small).}$$

  Examples: $\log n = o(n^k)$ for all $k > 0$, $(\log n)^{100} = o(\sqrt{n})$, $n \log n = o(n^2)$, $n = o(n \log \log n)$.

- **Exponentials:**

  These are functions for which $f(n) = c^n$ for some $c > 1$. We have:

  $$\frac{n^k}{c^n} \to 0 \quad \text{for any } k > 0 \text{ (however large) and any } c > 1 \text{ (however small)}$$

  so any poly is $o(c^n)$ for any $c > 1$.
  Examples: $n^{10000000} = o(1.000000001^n)$, $2^n = o(3^n)$, $n^{100}2^n = o(3^n)$.

We sometimes use big-O notation as part of a more complicated mathematical expression. For example, $f(n) = 2^{O(n)}$ is shorthand for "there exist $c > 0$ and integer n_0 such that for all $n \geq n_0$, $f(n) \leq 2^{cn}$". One can even write (valid!) expressions like "$O(n) + O(n) = O(n)$".

Self Test Question:
Determine which of the following are true and which are false:
- $2n = O(n)$,
- $n^2 = O(n)$,
- $n^2 = O(n \log^2 n)$,
- $2^{2^n} = O(2^{2^n})$,
- $42 = O(1)$,
- $n = o(2n)$,
- $2n = o(n^2)$,
- $n = o(\frac{1}{n})$,
- $42 = o(1)$.

Solution:
- $2n = O(n)$, since there exists a constant $c = 2$ and $n_0 = 1$ s.t. $2n \leq 2 \cdot n$ for all $n \geq 1$.
- n^2 is not $O(n)$, since for any $c > 0$ and any n_0 you can find an $n \geq n_0$ s.t. $n^2 > c \cdot n$ (e.g. $n := n_0 + \lceil c \rceil$; alternatively, we could have checked that $\frac{f(n)}{g(n)} = \frac{n^2}{n} = n$ is not lower or equal to any constant for all large n).
- n^2 is not $O(n \log^2 n)$, since $\frac{n^2}{n \log^2 n} = \frac{n}{\log^2 n}$ cannot be bounded by any constant as we know that $\frac{\log^2 n}{n} \to 0$ (if for some c we had that: $\frac{n}{\log^2 n} \leq c$ for all sufficiently large n, then $\frac{\log^2 n}{n} \geq \frac{1}{c}$ for all sufficiently large n, contradicting the fact that it converges to 0.
- $2^{2^n} = O(2^{2^n})$, since for *any* f we have $f(n) = O(f(n))$, as there exists $c = 1$ and $n_0 = 1$ s.t. for all $n \geq n_0$: $\frac{f(n)}{f(n)} \leq c$.
- $42 = O(1)$, since for any constants c_1, c_2 we have $c_1 = O(c_2)$ (take $c := \frac{c_1}{c_2}$).
- n is not $o(2n)$, since $\frac{n}{2n} = \frac{1}{2} \not\longrightarrow 0$, as $n \to \infty$.
- $2n = o(n^2)$, since $\frac{2n}{n^2} = \frac{2}{n} \longrightarrow 0$.
- n is not $o(\frac{1}{n})$, since $\frac{1}{1/n} = n \not\to 0$.
- For any constants $c_1, c_2 > 0$, c_1 is never $o(c_2)$, since $\frac{c_1}{c_2} \not\to 0$ for $n \to \infty$.

Self Test Question:
The notation $f(n) = 2^{O(g(n))}$ means $f(n) \leq 2^{cg(n)}$ for some (fixed) c and all $n >$ some n_0.
Show that $3^n = 2^{O(n)}$ (Is $3^n = O(2^n)$?).
Also show that if $f(n) = 2^{O(\log n)}$ then $f(n) = O(n^c)$ for some c.

Solution:
For the first part we have that since $3^n = 2^{O(n)}$ means $3^n \leq 2^{c \cdot n}$ for some constant c. If we take the constant $c = 2$ then we see that $2^{c \cdot n} = 2^{2n} = 4^n \geq 3^n$. Thus this statement is true.
But $3^n = O(2^n)$ is false since $3^n/2^n = (3/2)^n$ which is not less than some constant as n increases since $3/2 \geq 1$.
For the final part note that if $f(n) = 2^{O(\log n)}$ then $f(n) \leq 2^{c \log n}$ for some c. But then notice that $2^{c \log n} = 2^{\log(n^c)} = n^c$. Hence $f(n)/n^c \leq 1$ as n grows, i.e. $f(n) = O(n^c)$.

Self Test Question:
Prove or disprove the following assertions.

1. If $f(n) = n^k$ and $g(n) = c^n$, for some constants $c > 1$, $k \geq 0$, then $f(n) = o(g(n))$.
2. If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.
3. $f(n) = 2^{\Theta(\log n)}$ if and only if $f(n) = \text{poly}(n)$.
4. If $f(n) = O(n)$ and $g(n) = \Theta(n)$, then:
   (a) $f(n) + g(n) = O(n)$;
   (b) $f(n)g(n) = O(n^2)$;
   (c) $f(n) - g(n) = O(1)$;
   (d) $\frac{f(n)}{g(n)} = O(1)$.

Solution:

1. Can be done by, for example, applying Taylor's theorem to e^n to obtain $e^n \geq n^{k+1}/(k+1)!$, and observing that $n^k = o(n^{k+1})$. This gives $n^k = o(e^n)$, hence $(n/\ln c)^k = o(c^n)$. As c is a constant greater than 1, the claim follows.
2. Not necessarily true. Take $f(n) = 2\log_2 n$, $g(n) = \log_2 n$. Then $2^{f(n)} = n^2$, $2^{g(n)} = n$, and $n = o(n^2)$.
3. True by explicitly writing out both sides.
4. (a) True. We have that there exist $c_1 > 0$, $c_2 > 0$, n_1, n_2 such that, for all $n \geq n_1$, $f(n) \leq c_1 n$, and for all $n \geq n_2$, $g(n) \leq c_2 n$. So, for all $n \geq \max\{n_1, n_2\}$, $f(n) + g(n) \leq (c_1 + c_2)n$.
   (b) True, similarly to the last part.
   (c) False; e.g. if $f(n) = 2n$, $g(n) = n$, $f(n) - g(n) = n$.
   (d) True. Note that we may have $g(n) = 0$ (for example) for small enough n, but for large enough n, $f(n)/g(n) \leq c$ for some constant c.

Self Test Question:
Let $T(n)$ be defined recursively by $T(1) = 1$, and $T(n) = cT(\lceil n/2 \rceil) + d$ for some constants $c, d > 1$. Show that $T(n) = \Theta(n^{\log_2 c})$.

Solution:
We can assume that n is a power of 2 without changing the asymptotics of $T(n)$. Then, by explicit calculation,

$$T(n) = c^{\log_2 n} + d(c^{\log_2 n - 1} + \cdots + c),$$

so $n^{\log_2 c} \leq T(n) \leq (d+1)n^{\log_2 c}$. As c and d are constant, this is $\Theta(n^{\log_2 c})$.

Self Test Question:
Give examples of functions f, g such that neither $f(n) = O(g(n))$, nor $f(n) = \Omega(g(n))$.

Solution:
Two simple examples are $f(n) = n$, $g(n) = n^2(1 + (-1)^n)$. Then there exist no $c > 0$, n_0 such that, for all $n \geq n_0$, $f(n) \leq cg(n)$; and similarly for $f(n) \geq cg(n)$.

0.4 Turing Machines

This section of notes recaps on the notion of Turing Machines from the first year. Note that there are many different (and essentially equivalent) ways of defining a Turing Machine. The following is a slight modification of what you saw in the first year.

The key idea of a Turing machine is that it allows us to theoretically model the idea of computation. For us a computation will be the execution of an *algorithm*. An algorithm is a method for solving a problem; more precisely, an effective method for calculating a function, expressed as a finite list of well-defined instructions. We will implement our algorithms using a basic, yet surprisingly powerful, kind of computer called a Turing machine.

A Turing machine can be thought of as a physical computational device consisting of a *tape* and a *head*. The head moves along the tape, scanning and modifying its contents according to the currently scanned symbol and its internal state. A "program" for a Turing machine specifies how it should operate at each step of its execution (i.e. in which direction it should move and how it should update the tape and its internal state). We assume that the tape is infinite in one direction and the head starts at the leftmost end.

Formally, a Turing machine M is specified by a triple (Σ, K, δ), where:

- Σ is a finite set of symbols, called the *alphabet* of M. We assume that Σ contains two special symbols $\square$ and $\triangleright$, called the *blank* and *start* symbols, respectively.
- K is a finite set of *states*. We assume that K contains a designated start state START, and a designated halting state HALT. K describes the set of possible "mental states" of M.

Figure 1: A typical configuration of a Turing machine, and the starting configuration on input 21.

- δ is a function such that $\delta : K \times \Sigma \to K \times \Sigma \times \{\leftarrow, -, \rightarrow\}$. δ describes the behaviour of M and is called the transition function of M.

A tape is an infinite sequence of cells, each of which holds an element of Σ. The tape initially contains $\triangleright$, followed by a finite string $x \in (\Sigma \backslash \{\square\})^k$ called the *input*, followed by an infinite number of blank cells $\square\square\ldots$. For brevity, in future we will often suppress the trailing infinite sequence of $\square$'s when we specify what the tape contains.

At any given time, the current state of the computation being performed by M is completely specified by a *configuration*. A configuration is given by a triple (ℓ, q, r), where $q \in K$ is a state and specifies the current "state of mind" of M, and $\ell, r \in \Sigma^*$ are strings, where ℓ describes the tape to the left of the head (including the current symbol scanned by the head, which is thus the last element in ℓ) and r describes the tape to the right of the head. Thus the initial configuration of M on input x is $(\triangleright, \text{START}, x)$.

At each step of M's operation, it updates its configuration according to δ. Assume that at a given step M is in state q, and the symbol currently being scanned by the head is σ. Then, if $\delta(q, \sigma) = (q', \sigma', d)$, where $d \in \{\leftarrow, -, \rightarrow\}$:

- the symbol on the tape at the position currently scanned by the head is replaced with σ';
- the state of M is replaced with q';
- if $d = \leftarrow$, the head moves one place to the left; if $d = -$, the head stays where it is; if $d = \rightarrow$, the head moves one place to the right.

We assume that, for all states p and q, if $\delta(p, \triangleright) = (q, \sigma, d)$ then $\sigma = \triangleright$ and $d \neq \leftarrow$. That is, the head never falls off the left end of the tape and never erases the start symbol. We also assume that $\delta(\text{HALT}, \sigma) = (\text{HALT}, \sigma, -)$. That is, when M is in the halt state, it can no longer modify its configuration. If M has halted, we call the contents of the tape the output (excluding the $\triangleright$ symbol on the left of the tape and the infinite string of $\square$'s on the right). If M halts with output y on input x, we write $M(x) = y$, and write $M : \Sigma^* \to \Sigma^*$ for the function computed by M. Of course, some machines M may not halt at all on certain inputs x, but just run forever. If M does not halt on input x, we can write $M(x) = \nearrow$ (in fact, we will never need to do so).

Example 1. *We describe a Turing machine M below which computes the function RMZ : $\{0,1\}^* \to \{0,1\}^*$, where RMZ(x) is equal to x with the rightmost 0 changed to a 1. If there*

are no 0's in x, the machine outputs x unchanged. M operates on alphabet $\{\triangleright, \square, 0, 1\}$, has states $\{START, TOEND, FINDZERO, HALT\}$, and has transition function δ which performs the following maps.

$$\begin{aligned}
(START, \triangleright) &\mapsto (TOEND, \triangleright, \rightarrow) \\
(TOEND, 0) &\mapsto (TOEND, 0, \rightarrow) \\
(TOEND, 1) &\mapsto (TOEND, 1, \rightarrow) \\
(TOEND, \square) &\mapsto (FINDZERO, \square, \leftarrow) \\
(FINDZERO, 0) &\mapsto (HALT, 1, -) \\
(FINDZERO, 1) &\mapsto (FINDZERO, 1, \leftarrow) \\
(FINDZERO, \triangleright) &\mapsto (HALT, \triangleright, \rightarrow)
\end{aligned}$$

For brevity, transitions that can never occur are not specified above; however, it should already be clear that writing programs in the Turing machine model directly is a tedious process. In future, we will usually describe algorithms in this model more informally.

Self Test Question:
Show that, the following variant of a Turing machine can be simulated efficiently by a standard Turing machine: A Turing machine defined in the same way as normal, but with a *two-way* infinite tape. The head starts at position 0 and can move arbitrarily far in either direction (by at most one position at each step).

Solution:
The basic idea is to fold the tape in two, so that the i'th cell of a one-way tape stores the contents of cells i and $-i$ of the two-way tape. This can be done using a new alphabet Σ^2. At each point in time the machine uses a head state to store whether it is in the negative part of the tape (in which case it operates on the first element of the pair) or the positive part (in which case it operates on the second element). See the proof of Claim 1.8 in Arora-Barak for more details (which is Claim 1.11 in the online draft).

Self Test Question:
Show that, the following variant of a Turing machine can be simulated efficiently by a standard Turing machine: A Turing machine with a 2-dimensional, one-way infinite "tape". The head starts at position $(0, 0)$ and can move arbitrarily far down or right. At each step, the head can move at most one position up, down, left or right.

Solution:
We split the two-dimensional tape into diagonal stripes, starting at the top left. Position (x, y) on the 2D tape corresponds to position $(x + y)(x + y + 1)/2 + y$ on the 1D tape. To simulate a move by one square in any direction to position (x', y'), the Turing machine calculates the new position using a work tape (which can be done efficiently).

0.4.1 Representation and countability

A Turing machine M's behaviour is completely specified by its transition function and the identity of the special states and symbols $\square$, $\triangleright$, START, HALT. Thus M can be represented by a finite sequence of integers, or (equivalently) a finite length bit-string. There are many reasonable ways that this representation can be implemented, and the details are not too important. Henceforth we simply assume that we have fixed one such representation method, and use the notation $\underline{M}$ for the bit-string representing M. The existence of such a representation implies that the set of Turing machines is countable, i.e. in bijection with $\mathbb{N}$.

It will be convenient later on to assume that every string $x \in \{0,1\}^*$ actually corresponds to some Turing machine. This can be done, for example, by associating each string $\underline{M}$ which is not a valid encoding of some Turing machine M with a single fixed machine, such as a machine which immediately halts, whatever the input.

0.5 Decision problems and languages

We will often be concerned with decision problems, i.e. problems with a yes/no answer. Such problems can be expressed naturally in terms of *languages*. A language is a subset of strings, $\mathcal{L} \subseteq \Sigma^*$. We say that $\mathcal{L}$ is *trivial* if either $\mathcal{L} = \emptyset$, or $\mathcal{L} = \Sigma^*$. Let M be a Turing machine. For each input x, if $M(x) = 1$, we say that M accepts x; if M halts on input x and $M(x) \neq 1$, we say that M rejects x. We say that M *decides* $\mathcal{L}$ if it computes the function $f_{\mathcal{L}} : \Sigma^* \to \{0,1\}$, where $f_{\mathcal{L}}(x) = 1$ if $x \in \mathcal{L}$, and $f_{\mathcal{L}}(x) = 0$ if $x \notin \mathcal{L}$. If there exists such an M then we say that $\mathcal{L}$ is decidable. On the other hand, we say that $\mathcal{L}$ is *undecidable* if $f_{\mathcal{L}}$ is uncomputable. Theorem 6 in the supplementary notes therefore says that the language

$$\textsc{Halt} = \{(\underline{M}, x) : M \text{ is a Turing machine that halts on input } x\}$$

is undecidable. We also say that M *recognises* $\mathcal{L}$ if:

- $M(x) = 1$ for all $x \in \mathcal{L}$;
- for all $x \notin \mathcal{L}$, either M halts with output $\neq 1$, or M does not halt.

Thus all decidable languages are recognisable, but the converse is not necessarily true.

Self Test Question:
Give an "implementation-level" description (i.e., without specifying the transition function, but describing how the Turing machine works) of a TM which decides the following language:

$$\{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ contains twice as many } \mathtt{0}\text{'s as } \mathtt{1}\text{'s}\} \ .$$

Solution:
We build a Turing machine (TM) M which decides the following language:

$$\{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ contains twice as many } \mathtt{0}\text{'s as } \mathtt{1}\text{'s}\} \ .$$

This exercise also serves to illustrate what we mean by an "implementation-level" description, that is: describe how the TM works, however, without detailing states and transition function.

M = "On input string w:

1. Scan the tape and find the first `1` that has not been marked.
   – if no unmarked `1` is found then go to Stage 5.
   – otherwise mark the found `1` and move the head back to the front of the tape.
2. Scan the tape and mark the first unmarked `0`. If no unmarked `0` is found then *reject*.
3. Scan the rest of the tape and mark the next unmarked `0`; if non is found then *reject*.
4. Move the head to the front of the tape and go to Stage 1.
5. Move the head to the front of the tape. Scan the tape to see if any unmarked `0`'s remain. If none are found then *accept*; otherwise, *reject*. "

Self Test Question:
Solve the following problems by giving "high-level" descriptions (use pseudo code) of Turing machines.

(i) Let L_1 and L_2 be decidable languages. Show that the union $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$ is also decidable.

(ii) Let L_1 and L_2 be recognisable languages. Show that the union $L_1 \cup L_2$ is also recognisable.

(iii) Let L_1 and L_2 be decidable languages. Show that the intersection $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ and } w \in L_2\}$ is also decidable.

(iv) Let L_1 and L_2 be recognisable languages. Show that the intersection $L_1 \cap L_2$ is also recognisable.

Solution:
(i) If L_1 and L_2 are decidable languages then, by definition, there exist TMs M_1 and M_2 such that for any input w, M_i halts with *accept* iff $w \in L_i$ and halts with *reject* iff $w \notin L_i$.
We construct M which recognises $L_1 \cup L_2$:

M = "On input w:
1. Run M_1 on w. If it accepts then *accept*.
2. Run M_2 on w. If it accepts then *accept*; otherwise *reject*. "

If either M_1 or M_2 accept then M accepts; if both reject then M rejects too.

(ii) The case of recognisability is a bit trickier: what if M_2 accepts w, but machine M_1 loops on input w? Then $w \in L_1 \cup L_2$, but M as constructed above would not accept w. We therefore have to run M_1 and M_2 "in parallel". M' thus simulates one step of machine M_1 then one step of machine M_2, and so on.

M' = "On input w:
1. Run M_1 and M_2 alternatively on w step by step. If either accepts then *accept*. If both halt and reject then *reject*. "

(iii) and (iv) These follow in much the same way.

Self Test Question:
Let A be the language containing only one string s, where

$$s = \begin{cases} 0 & \text{if God does not exist} \\ 1 & \text{if God exists} \end{cases}$$

Is A decidable? Why or why not? (Note: the answer does not depend on your beliefs.)

Solution:
Whether God exists or not, the language A is one of the following two languages: $A_0 = \{0\}$ or $A_1 = \{1\}$. Both of these languages are clearly decidable; A—whichever one it is—is therefore also decidable. We just don't know *which* one of the two languages it is; the language is decidable, while the choice of language might not be.

Self Test Question:
The *language* of a TM M is $L(M) := \{w \mid M \text{ accepts } w\}$. Define

$$E_{\text{TM}} := \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\} \ ,$$

that is, the set of all Turing machines which do not accept any input. Show that E_{TM} is undecidable. (Hint: Define $A_{\text{TM}} := \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$. Then reduce A_{TM} to E_{TM}; assume a TM R decides E_{TM}, then given $\langle M, w \rangle$, construct a machine M' and run R on $\langle M' \rangle$.)

Solution:
We show that $E_{\text{TM}} := \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable by **reducing** E_{TM} to $A_{\text{TM}} := \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$. This means, we assume that there exists a TM R which decides E_{TM} and use R to construct a TM S which decides A_{TM}.
A_{TM} is known to be **undecidable** (halting problem), thus no such machine S can exist. Therefore, also no TM R can exists which decides E_{TM}. E_{TM} is thus also undecidable.
Assume R decides E_{TM}, that is, given $\langle M \rangle$, R accepts iff M does not accept any input. Our task is to construct S which decides A_{TM}, that is, given $\langle M, w \rangle$, S should accept iff M accepts w.
The idea is to transform M into another TM M' which accepts w iff M accepts w, but which does not accept any other input. We can do so as follows:

$M' =$ "On input x:
1. If $x \neq w$ then *reject*
2. If $x = w$ then run M on w and *accept* if M accepts and *reject* otherwise."

We thus have:

$$L(M') = \begin{cases} \{w\} & \text{if } M \text{ accepts } w \\ \emptyset & \text{otherwise} \end{cases}$$

that is, M accepts $w \iff L(M') \neq \emptyset$. We can thus use the information whether M''s language is empty in order to decide whether the original TM M accepts w. We conclude by giving S which decides A_{TM}:

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w:
1. Use the description of M and w to construct the TM M' described above.
2. Run R on input $\langle M_1 \rangle$.
3. If R accepts then *reject*, if R rejects then *accept*. "

If R were a decider for E_{TM} then S would be a decider for A_{TM}. Since the latter cannot exist, we know that E_{TM} must be undecidable.

Self Test Question:
Define $EQ_{\text{TM}} := \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$. Show that EQ_{TM} is undecidable. (Hint: Reduce E_{TM} to EQ_{TM}.)

Solution:
We show that $EQ_{\text{TM}} := \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable by reducing E_{TM} to EQ_{TM}.
Suppose R decides EQ_{TM}. Then we could construct S deciding E_{TM} as follows:

$S =$ "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:
1. Construct a TM M_e which rejects all inputs.
2. Run R on input $\langle M, M_e\rangle$.
3. If R accepts then *accept*, if R rejects then *reject*. "

If R were a decider for EQ_{TM} then S would be a decider for E_{TM}. Since, by (i) we know that the latter cannot exist, we conclude that EQ_{TM} must be undecidable.

Self Test Question:
For the purposes of this question, say that a variant of the Turing machine is *equivalent* to the standard Turing machine if the set of languages $\mathcal{L} \subseteq \{0,1\}^*$ that can be decided by the variant model is the same as the set of languages that can be decided by the standard model. Is each of the following variants of the Turing machine equivalent to the standard Turing machine?

1. A Turing machine which operates on a two-way infinite tape, and can move arbitrarily far in each direction. That is, the transition function is of the form $\delta : K \times \Sigma \to K \times \Sigma \times \mathbb{Z}$, where the last integer specifies how far to move to the right (if positive), or left (if negative).
2. A Turing machine operating on the infinite alphabet $\Sigma = \mathbb{N}$.
3. A Turing machine which cannot stay still. That is, the transition function is of the form $\delta : K \times \Sigma \to K \times \Sigma \times \{\leftarrow, \rightarrow\}$.
4. A Turing machine which cannot move left. That is, the transition function is of the form $\delta : K \times \Sigma \to K \times \Sigma \times \{-, \rightarrow\}$

Solution:

1. This machine can be simulated by a standard Turing machine M with two two-way infinite tapes as follows. When a transition a certain number of steps to the left or right should be made, M writes the number of steps desired to the second tape and changes to a "keep moving left" (or "keep moving right") state. M then alternates moving to the left (or right) and decrementing the counter until it reaches 0.

2. This is not equivalent to the standard Turing machine. It can decide any language $\mathcal{L}$ – even if $\mathcal{L}$ is undecidable – by reading the input x and storing it as an integer on the tape, then having a state in which each integer is mapped to 1 or 0 depending on whether $x \in \mathcal{L}$.

3. This machine can simulate a standard Turing machine by having a special "stay still" state, and moving to the right followed by the left when it goes into this state

4. This is not as powerful as the standard Turing machine. We show this by giving a language $\mathcal{L}$ that can be decided by a standard Turing machine, but not by a Turing machine which cannot move left. Define

$$\mathcal{L} = \{yy : y \in \{0,1\}^n, n \in \mathbb{N}\};$$

that is, $\mathcal{L}$ is the language of strings which are the concatenations of two equal strings. Let the input x be a $2n$-bit string made up of two n-bit strings, for some n. Now consider a Turing machine M which never moves left. The machine has to determine whether the first half of x is equal to the second half. At some point, it must move from position n to position $n+1$ (or, for any $\ell \in \{0,1\}^n$, it could not distinguish $\ell\ell$ from ℓr for any $r \neq \ell$). M only has a finite number of states. Therefore, for large enough n, there must exist strings ℓ, $\ell' \neq \ell$ such that M is in the same state when it moves from position n to position $n+1$. Thus, M must perform the same computation on both the pairs $\ell\ell$ and $\ell'\ell$. As M should accept the first, and reject the second, it must be wrong on at least one.

Self Test Question:
Let $\mathcal{L}$ be a language such that $|\mathcal{L}|$ is finite. Show that $\mathcal{L}$ is decidable.

Solution:
The following Turing machine M decides $\mathcal{L}$. M has $|y|$ states for each possible input $y \in \mathcal{L}$. For each such y, the machine reads the input x and outputs 1 if $x = y$ (which can be done by checking each bit of x in turn), otherwise continuing to the next $y' \in \mathcal{L}$. If $x \neq y$ for all $y \in \mathcal{L}$, M outputs 0.

Self Test Question:
Is HALT recognisable? Let $\overline{\text{HALT}}$ be the language defined by

$$\overline{\text{HALT}} = \{(\underline{M}, x) : M \text{ is a Turing machine that does not halt on input } x \}.$$

Is $\overline{\text{HALT}}$ recognisable?

Solution:
HALT is recognisable: simulate M and accept if M halts. But $\overline{\text{HALT}}$ is not recognisable. If it were, then we could decide HALT by alternately simulating one step of the recogniser for HALT and the recogniser for $\overline{\text{HALT}}$.

Self Test Question:
For any Turing machine M, let $L(M)$ denote the language recognised by M. Let S be a set of languages, and let $\mathcal{L}_S$ be the language

$$\mathcal{L}_S = \{\underline{M} : L(M) \in S\}.$$

That is, $\mathcal{L}_S$ is the set of Turing machines that recognise languages in S. Show that $\mathcal{L}_S$ is either trivial or undecidable. Conclude that the language

$$\mathcal{L}_k = \{\underline{M} : |L(M)| = k\}$$

is undecidable for any k. In other words, for any k it is undecidable whether, given a Turing machine M, M accepts exactly k inputs.

Solution:
Suppose for a contradiction that $\mathcal{L}_S$ is non-trivial and decidable, and suppose that $\emptyset \notin S$ (otherwise, replace S with $\overline{S}$; we can do this because $\mathcal{L}_S$ is decidable if and only if $\mathcal{L}_{\overline{S}}$ is decidable). We will show that we can now determine if an arbitrary machine M accepts a given input w, which (as we know) is undecidable, implying the desired contradiction.
Consider a machine N which takes input x. N first simulates M on input w (ignoring x for now). If M does not halt, neither does N. If M rejects w, N rejects. If M accepts w, N simulates M' on input x, for some arbitrary M' such that $\underline{M'} \in \mathcal{L}_S$.
We now claim that $\underline{N} \in \mathcal{L}_S$ if and only if M accepts w. If M does not accept w, N does not accept any inputs, so $L(N) = \emptyset \notin S$. And if M accepts w, N behaves like M' and hence accepts x if and only if $x \in \mathcal{L}_S$. Thus, given the ability to decide membership in $\mathcal{L}_S$, we can determine whether M accepts w.
This result is known as *Rice's Theorem*: "any non-trivial property of Turing machines is undecidable".
The conclusion follows because $\mathcal{L}_k$ is non-trivial; one can easily write down Turing machines which accept exactly k strings, for any k (for example, the machine which checks whether the input is a string of k 1s).

0.6 Polynomial and Exponential Time Turing Machines

It is intuitively clear that some algorithms are more efficient than others; the Turing machine model allows us to formalise this notion. The first important resource which we will consider is time. Let $f : \Sigma^* \to \Sigma^*$, $T : \mathbb{N} \to \mathbb{N}$ be functions. If M is a Turing machine with alphabet Σ, we say that M computes f in time $T(n)$ if, for all $x \in \Sigma^*$, M halts with output $f(x)$ using at most $T(|x|)$ steps. We stress that the "n" in $T(n)$ is used as a placeholder; there is no implication that $|x| = n$. In particular, we say M computes f in time poly(n) if, for all $x \in \Sigma^*$, M halts with output $f(x)$ in time poly$(|x|)$.

A *complexity class* is a family of languages We write the names of complexity classes in SANS SERIF, and usually write the names of languages in Small Caps. A major goal of computational complexity theory is to classify decision problems (i.e. languages; we will use the terms "decision problem" and "language" essentially interchangeably throughout) into complexity classes of similar levels of difficulty. Here are some examples of the sort of decision problems which we will consider.

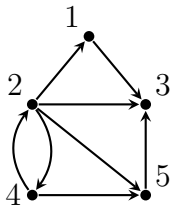
- Primes. Given an integer N specified as n binary digits, is N prime? Equivalently, decide the language Primes $= \{N \in \{0,1\}^* : \forall p, q \geq 2, N \neq p \times q\}$, where $\times$ is usual integer multiplication. Observe that we could have specified N in decimal without significantly changing the complexity of this problem.

- We will frequently be interested in problems associated with graphs. A graph $G = (V, E)$ is a set of n vertices V and m edges $E \subseteq V \times V$. G is said to be undirected if $(i, j) \in E \Leftrightarrow (j, i) \in E$, and directed otherwise. We write $v \to w$ if there is an edge from v to w. One way to specify G is by its adjacency matrix: an $n \times n$ matrix A where $A_{ij} = 1$ if $(i, j) \in E$, and $A_{ij} = 0$ otherwise. Alternatively, G can be specified by an adjacency list, which associates with each vertex a list of integers identifying which its neighbours are.

  A particularly important graph problem is known as Path. We are given as input a graph G (in one of the above representations) and two vertices s and t. Our task is to decide whether there is a path in G from s to t, i.e. a sequence $s \to v_1 \to v_2 \to \cdots \to t$.

We could have defined Path as (for example) the language specified as the subset of $\{(A, s, t) : A \in \{0,1\}^*, s, t \in \{0,1\}^*\}$ such that $|A| = n^2$ for some integer n, $1 \leq s, t \leq n$, and there is a path from s to t in the graph corresponding to the adjacency matrix A, but this level of formality rapidly becomes tedious. In future, when we talk about problems of the form "given $x \in S$, determine whether x satisfies property P", we will assume that x is specified in a sensible manner, and that it is possible to easily determine whether $x \in S$.
We are now ready to define the first complexity class we will consider.

Definition 1. *Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A language $\mathcal{L} \subseteq \{0,1\}^*$ is said to be in* DTIME$(T(n))$ *if there is a multiple tape Turing machine that runs in time $cT(n)$ for some constant $c > 0$ and decides $\mathcal{L}$.*



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad \big(\{3\}, \{1,3,4,5\}, \{\}, \{2,5\}, \{3\}\big)$$

Figure 2: A directed graph and its representation as an adjacency matrix and an adjacency list.

The Linear Speedup Theorem justifies our choosing arbitrary $c > 0$ in this definition[4]. The "D" stands for "deterministic"; later on we will also study nondeterministic and randomised models.

We shall use the following theorem without proof, but see the books mentioned at the beginning of this lecture.

Theorem 1 (Time Hierarchy Theorem)**.** *If f and g are time-constructible[5] functions such that $f(n)\log f(n) = o(g(n))$, then*

$$\mathsf{DTIME}(f(n)) \subset \mathsf{DTIME}(g(n)).$$

We now come to an important concept, the complexity class P. This class is simply defined by

$$\mathsf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c).$$

Crucially, c does not grow with n. In words, P is the family of languages which can be decided by a Turing machine in polynomial time (i.e. time which is polynomial in the input size). This class will capture our notion of "efficient computation". Why is this a reasonable notion of efficiency? After all, an algorithm which runs in time $\Theta(n^{100})$ can hardly be said to be "efficient"; for some applications, even algorithms which run in time $\Theta(n^2)$ may be too slow. A simple empirical reason to like the class P is that in most cases (but not all!) where we have an algorithm for a problem which runs in polynomial time, the degree of the polynomial is in fact reasonable (say 3 or 4).

For any language $\mathcal{L} \subseteq \Sigma^*$ its complement $\mathcal{L}'$ is the language $\mathcal{L}' = \Sigma^* - \mathcal{L} = \{w : w \notin \mathcal{L}\}$. If A is any class of languages, co-A is the class of all complements of languages in A.

Self Test Question:
Show that $\mathsf{P} = \mathsf{co\text{-}P}$.

[4]See the Advanced Technical Notes for this chapter

[5]A technical term just meaning a function you or I would recognize as a function, again see the Advanced Technical Notes for this chapter.

Solution:
Since our definition of P is based on decidability, this is immediate.
If we used recognizability as our definition then we need to do a little more work: Let $L' \in \mathsf{co\text{-}P}$ then $L' = \Sigma^* - L$ for some $L \in \mathsf{P}$. Thus there is a TM M which when run on L will output accept in time n^c for some c. We define the following machine M' to recognize L' in polynomial time. We pass it the machine M if it returns accept then M' returns reject. If M runs for more than n^{c+1} steps then we terminate it and M' returns accept. Thus $\mathsf{co\text{-}P} \subseteq \mathsf{P}$. The other inclusion can be shown in the same way.

Self Test Question:
Show that P is closed under unions and intersections (i.e. if $\mathcal{L}_1, \mathcal{L}_2$ are in P then so is $\mathcal{L}_1 \cup \mathcal{L}_2$ and $\mathcal{L}_1 \cap \mathcal{L}_2$).
Show that P is closed under concatenations; $\mathcal{L}_1 \circ \mathcal{L}_2 = \{w : w = w_1 w_2$ for some $w_1 \in \mathcal{L}_1$ and $w_2 \in \mathcal{L}_2\}$.

Solution:
For unions and intersections this is just the fact that the sum of polynomial runtimes is a polynomial runtime.
For concaternations we need to look at the $n+1$ possible prefixes and test whether they are in $\mathcal{L}_1$, and ditto for the corresponding $n+1$ suffixes and the language $\mathcal{L}_2$. Thus the complexity of the algorithm will be $(n+1) \cdot (f_1(n) + f_2(n))$ where f_1 and f_2 are the complexities of the machine to decide membership of $\mathcal{L}_1$ and $\mathcal{L}_2$. If f_1 and f_2 are polynomials then so is $(n+1) \cdot (f_1(n) + f_2(n))$.

A class of languages which certainly does not correspond to a real-world notion of efficiency is

$$\mathsf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{n^c}),$$

the class of languages which can be decided in exponential time. In particular,

Theorem 2. $\mathsf{P} \subset \mathsf{EXP}$.

Proof. As $n^c = O(2^n)$ for any constant c, $\mathsf{DTIME}(n^c) \subseteq \mathsf{DTIME}(2^n)$. On the other hand, by the Time Hierarchy Theorem $\mathsf{DTIME}(2^n) \subset \mathsf{DTIME}((2^{2n+1})^3) \subseteq \mathsf{EXP}$. This last point establishes that $\mathsf{P} \neq \mathsf{EXP}$, i.e. it is a true set inclusion. □

To stress the vast gulf between polynomial and exponential time, consider the following thought experiment. Imagine we have two algorithms for some problem, the first of which runs in time n^2, the second in time 2^n, and we execute these algorithms on a computer which performs one elementary operation per microsecond (10^{-6} seconds). Then, on a problem instance of size 100, the first algorithm will complete in a hundredth of a second; however, after 40 quadrillion years we will still be waiting for the second algorithm to complete.

There are many problems for which it is easy to find an exponential-time algorithm, but it is far more challenging to find an algorithm which runs in polynomial time. A good example is provided by the problem INTEGER FACTORISATION: given an n-digit integer N and an integer K, both expressed in binary, does N have a prime factor smaller than K? There is an easy exponential-time algorithm for this problem: simply try every possible prime number $2 \leq j \leq K$, and see if j divides N. The most efficient algorithm currently known for INTEGER FACTORISATION runs in time $e^{O(n^{1/3}(\log n)^{2/3})}$ and is based on advanced number-theoretic ideas. However, it is not known whether there exists a polynomial-time algorithm for this problem.

Worst vs Average Case

A key point which you need to appreciate is that complexity theory is generally about *worst case* running times; i.e. the run time over all instances of a given input length. This can often be very different from the average run time across all instances. In some sense you have seen this before in sorting algorithms, (non-randomized, deterministic) quicksort takes $O(n^2)$ steps on worst case, but will sort almost all lists in time $O(n \cdot \log n)$.

Self Test Question:
It is believed that determining where a large integer N has a factor less than $M > 1$ is not in P. However, the average complexity is constant time, why?

Solution:
Fifty percent of the time the run time will require one operation (testing division by two), then one sixth of the time it will require two operations (testing division by two and division three) and so on. Thus the average run time will be, for what is called the method of trial division,

$$1 + \frac{2}{2} + \frac{3}{6} + \frac{4}{30} + \frac{5}{210} + \cdots \approx 2.66$$

assuming M is quite big, the smaller M is the average number of steps will be even smaller.

0.7 Supplementary Notes

This section is to provide background and additional information. For this lecture this is focused on the area of Turing Machines.

Alphabet

We immediately observe that we can fix the alphabet $\Sigma = \{\triangleright, \square, 0, 1\}$ (i.e. use a binary alphabet, with two additional special symbols) without restricting the model, because we can encode any more complicated alphabets in binary. Imagine that we have a machine M whose alphabet Σ contains

K symbols (other than $\triangleright$, $\square$). Encode the i'th symbol as a $k := \lceil \log_2 K \rceil$-bit binary number, i.e. a string of k 0's and 1's. Define a new Turing machine $\widetilde{M}$ which will simulate M as follows.

The tape on which $\widetilde{M}$ operates contains the binary encodings of the symbols of the tape on which M operates, in the same order. In order to simulate one step of M, the machine $\widetilde{M}$:

1. Reads the k bits from its tape which encode the current symbol of Σ which M is scanning, using its state to store what this current symbol is. If the bits read do not correspond to a symbol of Σ, $\widetilde{M}$ can just halt.

2. Uses M's transition function to decide what the next state of M should be, given this input symbol, and stores this information in its own state.

3. Updates the k bits on the tape to encode the new symbol which M would write to the tape.

This gives a very simple example of an encoding, i.e. a map which allows us to translate between different alphabets. Henceforth, we will often assume that our Turing machines operate on a binary alphabet. Similarly, we can assume that the states of the Turing machine are given by the integers $1, \ldots, m$, for some finite m.

Encoding of the input

We may sometimes wish to pass multiple inputs $x_1, \ldots, x_k$ to M. To do so, we can simply introduce a new symbol "," to the alphabet, enabling M to determine when one input ends and another begins.

Multiple-tape Turing machines

The Turing machine model is very simple and there are many ways in which it can (apparently!) be generalised. Remarkably, these "generalisations" usually turn out not to be any more powerful than the original model.

A natural example of such a generalisation is to give the machine access to multiple tapes. A k-tape Turing machine M is a machine equipped with k tapes and k heads. The input is provided on a designated input tape, and the output is written on a designated (separate) output tape. The input tape is usually considered to be read-only, i.e. M does not modify it during its operation. The remaining $k-2$ tapes are work tapes that can be written to and read from throughout M's operation. The work and output tapes are initially empty, apart from the start symbol. At each stage of the computation, M scans the tapes under each of its heads, and performs an action on each (modifying the tape under the head and moving to the left or right, or staying still). M's transition function is thus of the form $\delta : K \times \Sigma^k \to K \times (\Sigma \times \{\leftarrow, -, \rightarrow\})^k$. Observe that M's internal state is shared across tapes.

Theorem 3. *Given a description of any k-tape Turing machine M operating within $T(n)$ steps on inputs of length n, we can give a single tape Turing machine M' operating within $O(T(n)^2)$ steps such that $M'(x) = M(x)$ for all inputs x.*

Proof. Let the input x be of length n. The basic idea is that our machine M' will encode the k tapes of M within one tape by storing the j'th cell of the i'th tape of M at position $n+(j-1)k+i$. Thus the first tape is stored at $n+1, n+k+1, n+2k+1, \ldots$, etc. The alphabet of M' will be twice the size of the alphabet of M, containing two elements $a, \widehat{a}$ for each element a of the alphabet

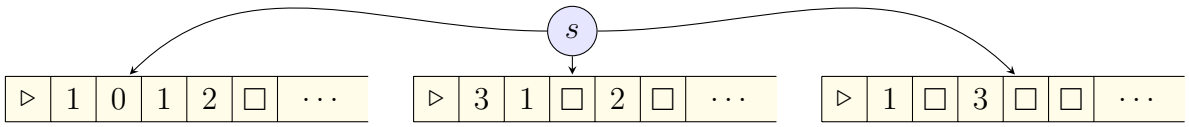


Figure 3: A typical configuration of a Turing machine with three tapes (input, work and output) on input 1012.

of M. A "hat" implies that there is a head at that position. Exactly one cell in the encoding of each tape will include an element with a hat.

To start with, M' copies its input into the correct positions to encode the input tape and initialises the first block $n+1, \ldots, n+k$ to $\widehat{\triangleright}$. This uses $O(n^2)$ steps (note that k is constant). Then, to simulate each step of M's computation, M' scans the tape from left to right to determine the symbols scanned by each of the k heads, which it stores in its internal state. Once M' knows these symbols, it can compute M's transition function and update the encodings of the head positions and tapes accordingly. As M operates within $T(n)$ steps, it never reaches more than $T(n)$ locations in each of its tapes, so simulating each step of M's computation takes at most $O(T(n))$ steps. When M halts, M' copies the contents of the output tape to the start of its tape and halts. As there are at most $T(n)$ steps of M to simulate, the overall simulation is within $O(T(n)^2)$ steps. □

We say that a model of computation is equivalent to the Turing machine model if it can simulate a Turing machine, and can also be simulated by a Turing machine. Thus the above theorem states that multiple-tape Turing machines are equivalent to single-tape Turing machines. The simple fact used in the proof that the amount of space used by a computation cannot be greater than the amount of time used will come up again later.

The universal Turing machine

The fact that Turing machines M can be represented as bit-strings $\underline{M}$ allows M to be given as input to another Turing machine U, raising the possibility of *universal* Turing machines: Turing machines which simulate the behaviour of any possible M. We now briefly describe (without going into the technical details) how such a simulation can be done.

Theorem 4. *There exists a two-tape Turing machine U such that $U(\underline{M}, x) = M(x)$ for all Turing machines M.*

Proof. As discussed above, we may assume that M has one tape. U's input tape stores a description of M (which is indeed fully specified by its transition function δ and identities of the special states START, HALT) and the input x. U's second tape stores the current configuration (ℓ, q, r) of M simply by concatenating the three elements of the configuration, separated by "," symbols. To simulate a step of M, U scans the input tape to find the new state, new symbol to be written and head movement to be performed. U then updates the second tape accordingly. If M halts, U cleans up the second tape to (ℓ, r) and then halts. □

The overloading of the notation M highlights the triple role of Turing machines: they are simultaneously machines, functions, and data!

Computability

It is not the case that all functions $f : \{0,1\}^* \to \{0,1\}$ can be computed by some Turing machine. Indeed, this follows from quite general arguments: as discussed, the set of Turing machines is in bijection with the countable set $\{0,1\}^*$, while the set of all functions $f : \{0,1\}^* \to \{0,1\}$ is in bijection with the uncountable set of all subsets of $\{0,1\}^*$. Explicitly, we have the following theorem.

Theorem 5. *There exists a function UC* $: \{0,1\}^* \to \{0,1\}$ *which is not computable by a Turing machine.*

Proof. Define UC as follows. For every $\underline{M} \in \{0,1\}^*$, if $M(\underline{M}) = 1$, then $\mathrm{UC}(\underline{M}) = 0$; otherwise (i.e. if $M(\underline{M}) \neq 1$ or M does not halt on input $\underline{M}$), $\mathrm{UC}(\underline{M}) = 1$. Assume towards a contradiction that UC is computable, so there exists a Turing machine N such that, for all $\underline{M} \in \{0,1\}^*$, $N(\underline{M}) = \mathrm{UC}(\underline{M})$. In particular, $N(\underline{N}) = \mathrm{UC}(\underline{N})$. But we have defined

$$\mathrm{UC}(\underline{N}) = 1 \Leftrightarrow N(\underline{N}) \neq 1,$$

so this is impossible. □

This sort of argument is known as diagonalisation, for the following reason. We write down an (infinite!) matrix A whose rows and columns are indexed by bit-strings $x, y \in \{0,1\}^*$, in lexicographic order. The rows are intended to correspond to (encodings of) Turing machines, and the columns correspond to inputs to Turing machines. Define $A_{xy} = 1$ if machine x halts with output 1 on input y, and $A_{xy} = 0$ otherwise. Then the function $\mathrm{UC}(x)$ is defined by negating the diagonal of this table. Since the rows represent all Turing machines, and for all x, $\mathrm{UC}(x)$ differs on the i'th input from the function computed by the i'th Turing machine, the function $\mathrm{UC}(x)$ cannot be computed for all x by a Turing machine. See Figure 4 for an illustration.

	0	1	00	01	...	$\underline{N}$
0	**0**	1	0	0		
1	0	**1**	0	1		
00	1	1	**1**	0		
01	0	1	0	**0**		
⋮					⋱	
$\underline{M}$						$1 - M(\underline{N})$

Figure 4: Diagonalisation for proving undecidability.

The function UC may appear fairly contrived. However, it turns out that some very natural functions are also not computable by Turing machines; the canonical example of this phenomenon is the so-called halting problem. The function $\mathrm{HALT}(\underline{M}, x)$ is defined as:

$$\mathrm{HALT}(\underline{M}, x) = \begin{cases} 1 & \text{if } M \text{ halts on input } x \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 6. *HALT is not computable by a Turing machine.*

Proof. Suppose for a contradiction that there exists a Turing machine M which computes HALT; we will show that this implies the existence of a machine M' which computes $\mathrm{UC}(\underline{N})$ for any N, contradicting Theorem 5. On input $\underline{N}$, M' computes $\mathrm{HALT}(\underline{N}, \underline{N})$. If the answer is 0 (i.e. N does not halt on input $\underline{N}$), M' outputs 1. Otherwise, M' simulates N on input $\underline{N}$ using Theorem 4, and outputs 0 if N's output would be 1, or 1 if N's output would not be 1. Note that this can be done in finite time because we know that N halts. □

This is our first example of a fundamental technique in computational complexity theory: proving hardness of some problem A by reducing some problem B, which is known to be hard, to solving problem A. We state without proof some other problems corresponding to functions which are now known to be uncomputable.

- Hilbert's tenth problem: given the description of a multivariate polynomial with integer coefficients, does it have an integer root? Hilbert's tenth problem was proposed in 1900 but only proven undecidable in 1970. An interesting survey of undecidability in various areas of mathematics has recently been produced by Poonen[6]. For more on the connections between undecidability and the foundations of mathematics, including Gödel's incompleteness theorem, see Papadimitriou chapter 6 or Sipser chapter 6.

- The Post correspondence problem. We are given a collection S of dominos, each containing two strings from some alphabet Σ (one on the top half of the domino, one on the bottom). For example,
$$S = \left\{ \left[\frac{a}{ab}\right], \left[\frac{b}{a}\right], \left[\frac{abc}{c}\right] \right\}.$$
The problem is to determine whether, by lining up dominos from S (with repetitions allowed) one can make the concatenated strings on the top of the dominos equal to the concatenated strings on the bottom. For example,
$$\left[\frac{a}{ab}\right] \left[\frac{b}{a}\right] \left[\frac{a}{ab}\right] \left[\frac{abc}{c}\right]$$
would be a valid solution.

- A *Wang tile* is a unit square with coloured edges. Given a set S of Wang tiles, determine whether tiles picked from S (without rotations or reflections) can be arranged edge-to-edge to tile the plane, such that abutting edges of adjacent tiles have the same colour. For example, the following set S does satisfy this property.
$$S = \left\{ \text{  }, \text{  }, \text{  }, \text{  } \right\}.$$

Could there be other "reasonable" models of computation beyond the Turing machine which can do things that the Turing machine cannot, such as solving the halting problem? Here "reasonable" should be taken to mean: "corresponding to computations we can perform in our physical universe". The Church-Turing Thesis says that this is not the case.

[6] `http://arxiv.org/pdf/1204.0299v1.pdf`

0.8 Advanced Technical Notes

The Entscheidungsproblem

Here we informally discuss a way in which the concrete-seeming Turing machine model can be used to attack problems in the foundations of mathematics itself. The forbidding-sounding "Entscheidungsproblem" (which is simply German for "decision problem") is the following question, first posed by Hilbert in 1928. Does there exist an algorithm which, given a set of axioms and a mathematical proposition, decides whether it is provable from the axioms? In other words, is the language of valid mathematical statements in a given axiomatic system decidable?

For example, consider statements about the natural numbers. We would like an algorithm which determines whether statements like the following are true:

$$\forall a, b, c, n[(a, b, c > 0 \wedge n > 2) \Rightarrow a^n + b^n \neq c^n].$$

We can define the alphabet of statements about the natural numbers as

$$\Sigma_{\mathbb{N}} := \{\wedge, \vee, \neg, (,), \forall, \exists, =, <, >, +, \times, 0, 1, x\},$$

where x denotes the possibility to have variables in our statements. In order to determine whether such statements are provable, we also need to choose a set of axioms. A standard set of axioms for the natural numbers is called Peano arithmetic; the details of these are not so important for the high-level discussion here.

Theorem 7. *The language of statements about the natural numbers provable from the axioms of Peano arithmetic is undecidable.*

Proof idea. Let M be a Turing machine and w be a bit-string. The idea is to construct a formula $\phi_{M,w}$ in the language of statements about the natural numbers that contains one free variable x, and such that $\exists x\, \phi_{M,w}$ is true if and only if M accepts w. x is intended to encode a computation history (i.e. a complete description of the operation of a Turing machine) as an integer, and the formula $\phi_{M,w}$ is designed to check whether x is a valid computation history for M on input w, corresponding to M accepting w. This checking can be performed using arithmetic operations $+$, $\times$. The details of this process are quite technical, but it should at least be plausible that such an encoding can be carried out; any possible configuration of M can be encoded as an integer, and given two configurations c_1, c_2, the constraint that M maps c_1 to c_2 can be enforced by "local" arithmetical checks, corresponding to the locality of Turing machines. □

Efficiency and Time-Bounded Computation

The Turing machine in Example 1 computes the function RMZ in time $2(n+1)$. Observe that this is a "worst-case" notion: for some inputs, this machine halts more quickly, but we are interested in its behaviour on the worst possible input. We first show that there is little point in calculating running times exactly for Turing machines, as one can essentially always tweak the machine to make it run faster.

Theorem 8 (Linear Speedup Theorem)**.** *For any function $f : \Sigma^* \to \Sigma^*$, if there is a k-tape Turing machine M which computes f in time $T(n) \geq n$, then for any $\epsilon > 0$ there is a k'-tape Turing machine N which computes f in time $\epsilon T(n) + n + 2$. If $k = 1$, then $k' = 2$; otherwise $k' = k$.*

Proof. For simplicity, assume in the proof that $k = 1$, $k' = 2$ (the general case is similar). We define a new Turing machine N with two tapes. N's alphabet contains, as well as every symbol in M's alphabet, a new symbol for each possible m-tuple of symbols in the alphabet of M, for some m. That is, if M had alphabet Σ, N has alphabet $\Sigma \cup \Sigma^m$. The idea will be to simulate m steps of M's operation using only one step of N. To start with, N reads its input tape. Whenever it reads a m-tuple $(\sigma_1, \ldots, \sigma_m)$, it writes the corresponding symbol in its extended alphabet to its work tape; if a $\Box$ is encountered partway through a m-tuple (meaning N has got to the end of the input), the symbol is padded by the right number of $\Box$'s. When N has finished reading the input x (which uses $|x| + 2$ steps), it returns the head of its work tape to the start (using a further $\lceil |x|/m \rceil$ steps). The work tape of N is henceforth treated as its input tape.

Now m steps of M can be simulated using 6 steps of N, as follows. First, N reads the cells under its head and the cells immediately to the left and right (by moving its head one step to the left, then two to the right, then one to the left). Dependent on the contents of these cells, N updates them according to m steps of M's transition function, using at most two more steps. This can be done because m steps of M cannot travel further than the current cell of N, or the cells immediately to the left and right; the updates made by m steps of M can only modify the current cell of N, or one of its two neighbours, so require at most two more steps to simulate.

The overall number of steps used to simulate M's computation on input x is thus at most $|x| + 2 + \lceil |x|/m \rceil + 6\lceil T(|x|)/m \rceil$. Recalling that $T(|x|) \geq |x|$ and taking $m = \lceil 7/\epsilon \rceil$ implies the claimed result. $\Box$

The restriction that $T(n) \geq n$ in the Linear Speedup Theorem is not very significant, as Turing machines running in time less than this bound are generally considered uninteresting because they cannot read all of their input. The proof of the theorem crucially used the fact that computation is *local* in the Turing machine model, i.e. the head can only affect the tape around its current position. This idea will occur again later.

By contrast with the Linear Speedup Theorem, it turns out that some problems really do require more time to be solved than others. That is, if we allow a Turing machine more time, it can solve more problems. In what follows we will need to restrict ourselves to so-called time-constructible functions for technical reasons;

Definition 2 (Time Constructable)**.** *$T : \mathbb{N} \to \mathbb{N}$ is said to be time-constructible if $T(n) \geq n$ and there is a multiple tape Turing machine M which computes the function $M(x) = T(|x|)$, $x \in \{0,1\}^*$, in time $O(T(n))$.*

Time-constructibility is not a very significant restriction, as most "natural" functions (e.g. polynomials, exponentials) are time-constructible.

Theorem 9 (Time Hierarchy Theorem, weak version)**.** *If $f(n)$ is time-constructible,* $\mathsf{DTIME}(f(n))$ *is strictly contained within* $\mathsf{DTIME}((f(2n+1))^3)$.

Proof. For any time-constructible f, consider the following language:

$$H_f = \{(\underline{M}, x) : M \text{ accepts } x \text{ in } f(|x|) \text{ steps}\}.$$

Here $\underline{M}$ is (the binary description of) a multiple tape Turing machine, and x is its input. H_f can be decided as follows. First, calculate $f(|x|)$ and write its binary representation onto a work tape. We will use this as a counter to determine when we need to stop simulating M. We now run

the universal Turing machine U to simulate M for $f(|x|)$ steps, requiring time at most $O(f(|x|)^3)$. The construction of U given in Theorem 4 would normally use at most $O(f(|x|)^2)$ steps for this simulation (because of the quadratic penalty we obtain from simulating M by a single tape Turing machine), but at each step we need to decrement the counter; $O(f(|x|))$ time is a generous upper bound to do this. We therefore have $H_f \in \mathsf{DTIME}(O(f(n)^3))$. By the Linear Speedup Theorem, this implies $H_f \in \mathsf{DTIME}(f(n)^3)$.

On the other hand, we will show that $H_f \notin \mathsf{DTIME}(f(\lfloor n/2 \rfloor))$ by a diagonalisation argument. Suppose towards a contradiction that there does exist a Turing machine M_{H_f} which decides H_f in $f(\lfloor n/2 \rfloor)$ steps. Then define a machine D_f which computes

$$D_f(\underline{M}) = \begin{cases} 0 & \text{if } M_{H_f}(\underline{M}, \underline{M}) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

Consider applying D_f to itself, i.e. computing $D_f(\underline{D_f})$, and assume that the answer is 0. This implies that $M_{H_f}(\underline{D_f}, \underline{D_f}) = 1$ and hence that D_f accepts $\underline{D_f}$, contradicting the assumption. Conversely, assume that the answer is 1. This implies that D_f does not accept $\underline{D_f}$ within $f(|\underline{D_f}|)$ steps. But for any M, $D_f(\underline{M})$ uses the same number of steps as M_{H_f} does on input $(\underline{M}, \underline{M})$, i.e. at most $f(\lfloor (2|\underline{M}| + 1)/2 \rfloor) = f(|\underline{M}|)$ steps. So if D_f accepts $\underline{D_f}$ it does so within $f(|\underline{D_f}|)$ steps. We have reached a contradiction.

Combining these two claims, we have $\mathsf{DTIME}(f(n)) \subset \mathsf{DTIME}((f(2n+1))^3)$. □

By using a more efficient universal Turing machine, one can improve this to the Time Hierarchy Theorem, Theorem 1.

Self Test Question:
Prove that $T(n) = 2^n$ is time-constructible.

Solution:
A simple way to do this is as follows. Have two tapes (input and output). First check if the input is empty. If so, write 0 to the output tape and halt. Otherwise, write a 1 to the output tape and move to the right. Then repeatedly check for whether the current cell in the input tape is empty; if so, halt. Otherwise, write 0 to the output tape and move to the right in both tapes. This computes $2^{|x|}$ in $O(|x|)$ steps, easily meeting the requirements of time-constructibility.

Self Test Question:
Give an example (not necessarily explicit) of a function $T : \mathbb{N} \to \mathbb{N}$ such that $T(n) \geq n$ and $T(n)$ is not time-constructible.

Solution:
For example, take $T(n) = n$ if the Turing machine described by writing n in binary halts, and $T(n) = n + 1$ otherwise. Then computing $T(n)$ would allow the halting problem to be solved, so $T(n)$ is uncomputable.

Lecture 1

The Problems PATH and CLIQUE and the Complexity Classes P and NP

1.1 Church Turing Thesis and The Feasability Thesis

The above is a *deterministic* Turing machine (DTM). Later we'll have other kinds: *non-deterministic* (NDTM), *probabilistic* (PTM) Turing machines etc.

TMs were invented by Alan Turing in 1936. The structure of the model can be motivated by "a human doing a calculation" – finite 'head' size but any amount of scrap working paper available. (Infiniteness of tape is not unreasonable: in any computation only some finite amount is used but this amount can be arbitrarily large as input size grows). The TM gives a precise mathematical formalisation of the intuitive notion of computation – a very powerful tool for the study of computability and its possibilities.

Are there other models of computation, and is TM a good formalisation? Yes: many other models of computation have been proposed/studied e.g. lambda calculus, recursion, random access machines, gate array model (which we'll see later) etc. Also we have many variants of the notion of TM itself e.g. TMs with tapes that are infinite to the left and right, TMs that can "stay put" as well as move L or R in a transition, TMs with multiple tapes, and more. All these models have their corresponding notion of what's *computable* and they all agree:

Thesis 1 (Church-Turing Thesis)**.** *Anything that can be computed, can be computed by a TM.*

The Turing machine was invented by Alan Turing in 1936, while a Fellow of King's College, Cambridge. His seminal paper "On Computable Numbers, with an Application to the Entscheidungsproblem" can easily be found online and is worth reading. Turing was actually not the first to prove that the Entscheidungs problem is unsolvable; Alonzo Church beat him by a few months, using a different model of computation (the so-called λ-calculus). In an appendix to Turing's paper, he proves that Church's model is in fact equivalent to the Turing machine, giving the first evidence for what is now called the Church-Turing thesis.

Related to this was the use by Kurt Gödel of a class of functions called *recursive functions*. The work of Church and Turing showed that the three types of function were equivalent.

The Church-Turing Thesis is a principle, not a theorem, because the first occurrence of "computed" is the intuitive notion. It should satisfy natural intuitive conditions e.g. that only a finite amount of work can be done in one step (so we cannot update an real number to full infinite precision etc.) Hence it is generally accepted that the TM model is a good model, properly capturing the notion of computability.

We will never actually want to write programs for TMs, but only use the TM formalism in some arguments demonstrating possibilities/limitations of computation and complexity generally. In view of the CT principle we can (and will) discuss algorithms at a higher level. Such descriptions could, if necessary, be reformulated in more basic TM terms but this generally extremely tedious and obscures the essential ideas.

TMs are all very well but what is computation *really*? It is "processing" of "information". What is "information"? It is always represented in *physical* degrees of freedom of a physical system (voltage levels, positions of switches etc) – a computer is always a *physical* device. Bit values 0 and 1 are just two distinguishable states of some physical system. What is processing? It is *physical evolution* of the system. Hence, possibilities and limitations of information storage, computation and how efficiently a computation can be carried out, all must depend on the *laws of physics* which characterise the allowable kinds of evolutions etc. (It cannot be derived from thought/mathematics alone). From this viewpoint, TMs are already a very *high* level characterisation of computation having sidestepped the fundamental issues of physics!

In fact, TMs capture the computational power of *classical* physics. (TMs are clearly implementable in classical physics and the converse statement is essentially a physics interpretation of the Church-Turing thesis). But what if we use some other (better, improved!) theory e.g. quantum physics, leading to a notion of "quantum computation"? It turns out that we get remarkable new modes of computation not available to TMs.

Returning to real world computers the question arises as to whether a Turing Machine approximates a real world computer; i.e. can real world computers compute everything that a TM can? A real world computer as we know it today is a *von Neumann machine* (stored program computer, with input/output and random access memory), theoretically this is very close to what is called a Random Access Memory Machine (RAM Machine), which is like a Turing Machine but any finite point on the tape can be read and written to in one step; as opposed to having to traverse the tape.

A key open problem in the early days of computer science was the following question

Thesis 2 (Feasability Thesis)**.** *Is a RAM machine "close" to a Turing Machine, not only in terms of what* can *be computed, but also in terms of* efficiency*.*

Surprisingly the answer to this thesis is Yes. In the conference STOC '84 it was shown that the Feasability Thesis is true in the sense that any function which can be computed on one can be computed on the other; and (more importantly) the drop in performance between a RAM machine and a standard TM is simply a polynomial function of the input size. In other words if we define complexity classes in terms of RAM machines (i.e. close to physical

computers) then the complexity class poly would agree with that for Turing Machines. What this means is that if we do not care about polynomial factors (which is essentially what we are going to do in this course) then a TM and a real computer are essentially the same.

1.2 The Problem PATH

There are many problems for which a naïve algorithm would run in exponential time, but which are nevertheless in P. We now give an example of this phenomenon, with other examples being given in the Supplementary Notes section for this lecture.

From now on, to avoid the painful prospect of calculating time complexities directly in the Turing machine model, we will take a more informal approach and calculate running time based on a model of an idealised "real computer". For example, we will assume that any symbol in the input can be accessed in time $O(1)$. As our main concern will be to distinguish polynomial-time algorithms from exponential-time algorithms, the details of this model are not so important; all that matters is that n steps of a computation in such a model can be simulated in $\text{poly}(n)$ time on a Turing machine.

Theorem 10. PATH *is in* P.

Here the input size is $O(n^2)$ bits, so we look for an algorithm which runs in time $\text{poly}(n^2) = \text{poly}(n)$. Before proving this theorem, we observe that there is a trivial *inefficient* algorithm for PATH: simply enumerate all valid paths of length up to n that start from vertex s and see if one of them ends up in t. However, there could be as many as n^n such paths, so this algorithm uses time super-polynomial in n.

Proof of Theorem 10. Recall that in the PATH problem we are given a directed graph $G = (V, E)$ and two nodes s and t, and have to decide if there is a path from s to t. Our algorithm will be based on a data structure known as a queue, which allows the operations of addition (to the end of the queue) and removal (from the start of the queue). The algorithm proceeds as follows.

1. Mark node s and add it to the queue.
2. While the queue is not empty:
   (a) Remove the first node y from the queue.
   (b) For all edges (y, u) such that u is not marked, mark u and add u to the end of the queue.
3. Accept if t is marked.

It is easy to see that this algorithm is correct; we now discuss its running time. A queue can be implemented such that enqueuing and dequeuing an element takes only time $O(1)$. Each node is only added to the queue at most once. Therefore, an upper bound on the

running time is given by the sum over all nodes y of the time required to check each of y's neighbours. If the input graph G has n vertices and m edges and is given in terms of an adjacency matrix, we get a bound of $O(n^2)$, and if G is given as an adjacency list we get a bound of only $O(n+m)$, as each edge is checked only once[1]. In either case, this is polynomial in the input size. □

The above algorithm is known as breadth-first search. Observe that the algorithm can easily be modified to actually output a path from s to t (if one exists), by maintaining a tree containing all nodes visited thus far. When we mark a node u neighbouring a node y, we add it to the tree as a leaf descending from y. Then, if we find t, we explore the tree in reverse, repeatedly visiting the parent of the current node until we get back to s. Reversing the direction of each of these arcs gives a path from s to t.

1.3 The Problem Clique

The problem Clique is defined as follows: Given an undirected graph G and an integer k, does G contain a clique on k vertices, i.e. a k-subset of the vertices such that every pair of vertices in the subset are connected by an edge?

This appears to be a very hard problem. The naive method of solving it would be to take every k-subset of vertices and then test whether these formed a clique. If we were unlucky and the input graph only had one clique and this was the last one to be evaluated then the run time of this algorithm would be

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k! \cdot (n-k)!} \approx \left(\frac{n}{k}\right)^k$$

where n is the number of vertices in the graph. Thus as the input size is k we see that the naive algorithm takes exponential time.

However, Clique is actually a very interesting problem in the following sense. Whilst it is hard to solve, if we have a solution then testing it is easy. Suppose you have a solution, i.e. a clique on k vertices, then testing it is a clique requires only $O(k^2)$ steps. We call this clique a *witness* or *proof* that the given graph lies in the language.

There is something quite fundemental going on here, which is at the heart of mathematics (and hence computer science), and indeed science. Coming up with the proof (or witness) is hard, whereas verifying it is easy. This is what research is about, it might be hard to come up with a new invention or result, but once you have discovered it then showing the solution is easy. Put another way, of more relevance to yourselfs, finding the solution to an exam question may be hard, but verifying a solution (i.e. marking) is usually easy. This notion of coming up with something being hard, but verifying a solution being easy arises in all sorts of life. And in computer science it forms the basis of many application areas; and the understanding of the relationship between coming up with a proof/witness and verifying the

[1]The meaning of big-O notation may not be clear in the case of expressions involving more than one independently growing parameter. Here, we tacitly understand m to be a function of n.

said proof/witness forms the cornerstone of complexity theory. This concept motivates the definition of the complexity class NP.

Self Test Question:
A k-clique in a graph $G = (V, E)$ is a set of k vertices in which every two vertices are connected by an edge. Show that

$$\text{TRIANGLE} := \{G \mid G \text{ is a graph with a 3-clique}\}$$

is in P (and generalise this to any other fixed value of k).
Note that CLIQUE $:= \{(G, k) \mid G$ is a graph with a k-clique$\}$ *is* NP*-complete!*

Solution:
Let G have n vertices. For any fixed k, there are

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} = O(n^k)$$

subsets of k vertices. To check if a given such subset is a k-clique, we need to check the presence of $(k-1) + (k-2) + \ldots + 1 = \frac{k(k-1)}{2}$ edges (from each of the k vertices to all $k-1$ others in the set). Hence to check *all* such subsets requires time $O(n^k)$. Now $\frac{k(k-1)}{2}$ look-up's in the adjacency matrix is $O(n^k)$ time if k is a *fixed* constant, i.e., polynomial time. Hence CLIQUE_k is in P.
Note that it is crucial that k is fixed and not part of the input, like for CLIQUE defined in the lecture. Since in the latter case, k is part of the input and thus n^k (with n being the number of vertices) is *not* a polynomial in the input length!

Self Test Question:
Let

$$\text{MODEXP} := \{(a, b, c, p) \mid a, b, c, p \text{ are integers such that } a^b \equiv c \pmod{p}\}.$$

Show that MODEXP $\in$ P. (Note that the most obvious algorithm does *not* run in polynomial time. Hint: How could you optimise this algorithm when b is a power of 2?)

Solution:
We can assume that that multiplication and the mod operation for integers can be done in polynomial time in their input length.[a] Let (a, b, c, p) be an instance of the MODEXP problem. Multiplying a by itself b times and then checking whether $a^b \equiv c \pmod{p}\}$ is not a good idea, since this will take time exponential in the input length (If a has n bits and b has m bits then a^b has in the order of $n \cdot 2^m$ bits). We thus use the *square-and-multiply* algorithm instead. That is, we compute $a^b \bmod p$ in a "clever" way and then compare it with c to decide membership in *MODEXP*. We construct the following TM M:

$M =$ "On input (a, b, c, p), a tuple of integers:
1. Let $b = b_1 \dots b_k$ be the bit representation of b.
2. Let $a' \leftarrow a \bmod p$; let $x \leftarrow 1$.
3. For $i = 1$ to k:
   If $b_i = 1$ then let $x \leftarrow x * a' \bmod p$.
   Let $x \leftarrow x^2 \bmod p$.
4. Check whether $x = (c \bmod p)$.

It should be clear that M correctly computes $x = a^b \bmod p$ and thus decides the language *MODEXP*. Let us analyse its running time. Let n be the input length, thus all of a, b, c and p have fewer than n bits.
This algorithm performs modular reductions, multiplications and squarings (which is a multiplication) which can all be done in polynomial time. The For-loop in 3. is executed less than n times (since k, the length of b, is less than n), thus 3. runs in polynomial time, as obviously do all other steps as well. M thus runs in polynomial time, which shows that $MODEXP \in \mathsf{P}$.

[a] A modular reduction of a number a by another number p can be implemented as a division with remainder, which requires $O(n)$ subtractions. Multiplication of two numbers of size $O(n)$ takes $O(n^2)$ time using schoolbook multiplication and results in a number of size n^2.

1.4 The Class NP

Definition 3. NP *is the class of languages* $\mathcal{L} \subseteq \{0,1\}^*$ *such that there exists a polynomial* $p : \mathbb{N} \to \mathbb{N}$ *and a polynomial-time Turing machine* M *such that, for all* $x \in \{0,1\}^*$, $x \in \mathcal{L}$ *if and only if there exists* $w \in \{0,1\}^{p(|x|)}$ *such that* $M(x, w) = 1$.

We call M a verifier for $\mathcal{L}$ and w a certificate or witness for x. To illustrate this definition, consider the following examples of languages / decision problems in NP.

- The Path problem is in NP. Indeed, any language $\mathcal{L} \in \mathsf{P}$ is automatically in NP; the verifier can simply ignore the certificate w and decide membership in $\mathcal{L}$ in polynomial time.

- The language Composites, defined to be the set of integers $\{N : N = p \times q, p, q \geq 2\}$, where N is expressed in binary. A certificate is the factorisation (p, q); given p and q, it can be checked in time polynomial in the input size (i.e. $\text{poly}(\log N)$) that $N = pq$. In fact, Composites is in P but this is not obvious!

- The problem INTEGER FACTORISATION: given two integers N, k, does N have a prime factor smaller than k? Once again, a certificate is the factorisation of N. However, in this case INTEGER FACTORISATION is not known to be in P.

Another, and very important, example of a problem in NP is known as boolean satisfiability (SAT). In order to discuss this, we pause to introduce some notation and concepts relating to boolean functions.

Boolean functions

- A boolean variable x_i takes values 0 or 1 (corresponding to "false" or "true").
- A boolean function $f : \{0,1\}^n \to \{0,1\}$ is a function of n boolean variables. The list of the values f takes on each input, written out in lexicographic order, is called the truth table of f.
- Boolean operations AND ($x_1 \wedge x_2$), OR ($x_1 \vee x_2$), and NOT ($\neg x$) are defined as one might expect: $x_1 \wedge x_2 = 1$ if and only if $x_1 = x_2 = 1$; $x_1 \vee x_2 = 1$ if either x_1, x_2 or both are 1; $\neg x = 1 - x$. We will also use the operations XOR (addition modulo 2, or $\oplus$) and logical implication ($\Rightarrow$); we summarise all of these below.

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

- De Morgan's laws state that

$$\neg(x_1 \wedge x_2 \wedge \cdots \wedge x_k) = (\neg x_1) \vee (\neg x_2) \vee \cdots \vee (\neg x_k), \text{ and}$$
$$\neg(x_1 \vee x_2 \vee \cdots \vee x_k) = (\neg x_1) \wedge (\neg x_2) \wedge \cdots \wedge (\neg x_k).$$

- A literal is a boolean variable, possibly preceded by a NOT; for example, x_1 and $\neg x_1$ are literals.
- A boolean formula is an expression containing boolean variables and AND, OR and NOT operations, such as $\phi(x_1, x_2, x_3) = x_1 \wedge (\neg(\neg x_2 \vee x_3) \wedge x_3)$.
- A boolean formula is in Conjunctive Normal Form (CNF) if it is written as $c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where each c_i is a *clause* which is the OR of one or more literals (e.g. the formula

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

  is in CNF). Similarly, a formula is in Disjunctive Normal Form (DNF) if it is the OR of clauses, each of which is the AND of one or more literals.

- A boolean formula is *satisfiable* if there is an assignment to the variables that makes it evaluate to true. For example, the previous formula is satisfiable (e.g. set $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$).

We have the following "universality" result for boolean formulae.

Theorem 11. *Any boolean function $f : \{0,1\}^n \to \{0,1\}$ can be expressed as a boolean formula ϕ_f in CNF.*

Proof. For any boolean function f, we can generate a boolean formula ϕ_f which represents f as follows. For each input $y \in \{0,1\}^n$ such that $f(y) = 1$, write down the clause $c_y = (\ell_1 \wedge \ell_2 \wedge \ldots \ell_n)$, where $\ell_i = x_i$ if $y_i = 1$, and $\ell_i = \neg x_i$ otherwise. Then set $\phi_f(x) = \bigvee_{y,f(y)=1} c_y(x)$. As $c_y(x)$ evaluates to true if and only if $x = y$, it is clear that, for all x, $\phi_f(x) = f(x)$. This procedure produces a formula in DNF. To obtain a CNF formula, take the negation of the DNF formula $\phi_{\neg f}$ and use De Morgan's law. □

We are now equipped to define SAT, which is the following problem:

Definition 4 (SAT Problem)**.** *Given a boolean formula ϕ in CNF, is ϕ satisfiable?*

To check that SAT $\in \mathsf{NP}$, observe that if we are given an assignment to the variables of ϕ which is claimed to make ϕ evaluate to true, we can just plug it in and check. However, it is not clear how to (efficiently) *find* an assignment that makes ϕ true. One can iterate over all possible assigments to the variables of ϕ; however, if ϕ depends on n variables, this takes time $\Omega(2^n)$, which is exponential in the input size (assuming that the number of clauses is $O(n)$).

We have the following straightforward containments.

Theorem 12. $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$.

Proof. For the first inclusion, suppose that $\mathcal{L} \in \mathsf{P}$, so there exists a polynomial-time Turing machine N which decides $\mathcal{L}$. Then, given an empty certificate w, the verifier can simply use N to decide $\mathcal{L}$. For the second inclusion, if $\mathcal{L} \in \mathsf{NP}$, we can decide $\mathcal{L}$ in time $2^{O(p(n))}$ by trying all possible certificates $w \in \{0,1\}^{p(n)}$ as inputs to the verifier M in turn. As $p(n) = O(n^c)$ for some $c > 1$, $\mathcal{L} \in \mathsf{EXP}$. □

Perhaps surprisingly, for some problems in NP, no algorithm is known that runs significantly faster than this naïve exponential-time algorithm.

Self Test Question:
Given languages $\mathcal{L}_1, \mathcal{L}_2 \in \mathsf{NP}$, prove that $\mathcal{L}_1 \cap \mathcal{L}_2 \in \mathsf{NP}$, $\mathcal{L}_1 \cup \mathcal{L}_2 \in \mathsf{NP}$.

Solution:
In the first case, we check both witnesses and accept if they are both correct; in the second case, we accept if either one is correct.

Self Test Question:
Given languages $\mathcal{L}_1 \subseteq \mathcal{L}_2$ such that $\mathcal{L}_1 \in \mathsf{NP}$, $\mathcal{L}_2 \in \mathsf{NP}$, is $\mathcal{L}_2 \backslash \mathcal{L}_1 \in \mathsf{NP}$?

Solution:
Not necessarily. Set $\mathcal{L}_2 = \{0,1\}^*$, $\mathcal{L}_1 = \text{SAT}$. Then $\mathcal{L}_2 \backslash \mathcal{L}_1 \in \mathsf{NP}$ if and only if $\mathsf{NP} = \mathsf{co\text{-}NP}$.

1.5 NP-Completeness

An important tool for us to understand which problems are in P and which are in NP, is the notion of polynomial-time reductions[2]. We say that $f : \{0,1\}^* \to \{0,1\}^*$ is a *polynomial-time computable function* if there is a Turing machine that, when started with input x on the tape, halts in time poly($|x|$) with output $f(x)$. By analogy with P, FP is defined to be the set of all polynomial-time computable functions.

We then say that the language $\mathcal{A}$ is polynomial time reducible to $\mathcal{B}$ if there is a polynomial-time computable function f such we can transform inputs to problem $\mathcal{A}$ into inputs to problem $\mathcal{B}$ such that the resulting outputs are the same. We write $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$.

It is clear from these definitions that, if $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$ and $\mathcal{B} \in \mathsf{P}$, then $\mathcal{A} \in \mathsf{P}$. In other words we can turn a poly-time algorithm for deciding $\mathcal{B}$ into a poly-time algorithm for deciding $\mathcal{A}$. This is done as follows: Suppose we are given an input problem $w \in \mathcal{A}$, we now turn it into an input problem for $\mathcal{B}$ by evaluating $f(w)$, which will take polynomial time. We now feed $f(w)$ into our algorithm to solve $\mathcal{B}$, and since the two outputs are the same we have solved the problem instance in $\mathcal{A}$. In some sense the $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$ means that problem $\mathcal{A}$ is no harder than problem $\mathcal{B}$ if we ignore polynomial factors in the relative complexities.

The above type of reduction is called a *Karp* reduction after Richard Karp. Another form of (almost equivalent but not quite) reduction is called a Turing or Cook reduction after Stephen Cook. In this latter form of reduction we have a poly-time algorithm that solves the problem instance in $\mathcal{A}$ using a polynomial number of calls to an algorithm which solves problem instances in $\mathcal{B}$.

A nice aspect of polynomial-time reductions is that they compose: if $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$, and $\mathcal{B} \leq_{\mathsf{P}} \mathcal{C}$, then $\mathcal{A} \leq_{\mathsf{P}} \mathcal{C}$. This is another major motivation for choosing P as our class of "reasonable" problems.

Self Test Question:
Prove the claim that polynomial-time reductions compose: if $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$, and $\mathcal{B} \leq_{\mathsf{P}} \mathcal{C}$, then $\mathcal{A} \leq_{\mathsf{P}} \mathcal{C}$.

[2]These reductions also form the heart of modern cryptography, so their application is way beyond complexity theory and reaches out into highly practical matters

Solution:
We are given that there exist polynomial-time computable functions f and g such that $x \in \mathcal{A}$ if and only if $f(x) \in \mathcal{B}$, and $y \in \mathcal{B}$ if and only if $g(y) \in \mathcal{C}$. We need to show that there is a polynomial-time computable function h such that $x \in \mathcal{A}$ if and only if $h(x) \in \mathcal{C}$; it suffices to show that the composition $g \circ f$ is polynomial-time computable. The Turing machine which achieves this simply computes $f(x)$ (in polynomial time), then computes $g(f(x))$.

The concept of NP-completeness is a way of formalising the idea of what the "hardest" problems in NP are. To define it we use the fact that we do not care about polynomial-time reductions between problems.

Definition 5 (NP-Hardness and Completeness)**.**

- *We say that a language $\mathcal{A}$ is* NP*-hard if, for all $\mathcal{B} \in \mathsf{NP}$, $\mathcal{B} \leq_{\mathsf{P}} \mathcal{A}$.*
- *We say that a language $\mathcal{A}$ is* NP*-complete if $\mathcal{A} \in \mathsf{NP}$, and $\mathcal{A}$ is* NP*-hard.*

So, if $\mathcal{A}$ is NP-hard, deciding membership in $\mathcal{A}$ is, up to polynomial terms in the running time, at least as hard as the hardest problem in NP. This can be interpreted as evidence that there is no polynomial-time algorithm for $\mathcal{A}$, because if this were true, there would be a polynomial-time algorithm for every problem in NP, and we would have $\mathsf{P} = \mathsf{NP}$. The probability of this being the case is debatable, but it is generally considered very unlikely. Soon we will see some substantial evidence that $\mathsf{P} \neq \mathsf{NP}$, at least if one is a pessimist; the evidence being that a number of important problems turn out to be NP-complete.

However, it is perhaps not obvious that there should exist *any* NP-complete problems, let alone any "natural" ones. The remarkable fact that such problems do exist is called the Cook-Levin Theorem.

Theorem 13 (Cook-Levin Theorem)**.** *SAT is* NP*-complete.*

In order to prove this theorem, we will need to understand another, equivalent, definition of the class NP, as the class of languages recognised by a *nondeterministic* Turing machine. In fact, this was the original way in which NP was defined, and NP stands for "nondeterministic polynomial time" (**not** "non-polynomial time").

Self Test Question:
Recall that $\mathsf{EXP} = \bigcup_k \mathrm{TIME}(2^{n^k})$ and that $\mathsf{NP} \subseteq \mathsf{EXP}$, also recall a language L is NP*-hard* (resp. EXP-hard) if every language in NP (resp. EXP) is polynomial-time reducible to L. (Thus if $L \in \mathsf{NP}$ is NP-hard then L is NP-complete.) Show that if every NP-hard language is EXP-hard then $\mathsf{EXP} = \mathsf{NP}$.

Solution:
We will use the following two facts from the lectures: By the Cook-Levin theorem SAT is NP-complete (which, by definition means that SAT$\in$ NP and SAT is NP-hard). Moreover, (since we can simulate every polynomial-time non-deterministic TM by an exponential-time deterministic TM) we have NP $\subseteq$ EXP. We thus only need to show EXP $\subseteq$ NP.
SAT is NP-hard, thus by the questions assumption, it is also EXP-hard. This means that for any language $L \in$ EXP: $L \leq_p SAT$. Together with $SAT \in$ NP, yields $L \in$ NP. Thus EXP $\subseteq$ NP.

Self Test Question:
Let Unary Primes be the following language.

$$\text{Unary Primes} = \{1^n : n \text{ is prime}\}.$$

Prove that, if Unary Primes is NP-complete, then P = NP.

Solution:
It suffices to show that Unary Primes is in P, i.e. to give a poly(n) time algorithm to determine whether n is prime. To do this, we can simply try every number between 2 and $\sqrt{n}$ for being a factor of n, which takes time poly(n).

Self Test Question:
Prove that $\emptyset$ is not NP-complete.

Solution:
If $\emptyset$ were NP-complete, there would exist a reduction f from SAT to $\emptyset$. That is, given a boolean formula ϕ, we would have $f(\phi) \in \emptyset$ if and only if ϕ is satisfiable. But, as $\emptyset$ is empty, no such reduction can exist.

1.6 Supplementary Notes

Computing Edit Distance is in P

In this example we will consider is the problem of calculating edit distance between strings, an important problem in bioinformatics and elsewhere. Given two strings $x, y \in \Sigma^*$, the edit distance ED(x, y) is the number of edits required to change x into y, where an edit consists of removing, inserting or replacing a symbol in x. For example, ED$(baca, abba) = 3$ via the sequence of edits

$$baca \mapsto baba \mapsto aba \mapsto abba.$$

For any given pair of strings there are infinitely many sequences of edits which can transform one into the other. However, it turns out that we can find a shortest such sequence very efficiently.

Theorem 14. *Given two strings x, y of length at most n, $ED(x, y)$ can be computed in time $O(n^2)$.*

Proof. The technique we will use is known as dynamic programming, which is an important tool in algorithm design. Roughly speaking, the method works by breaking a problem down into simpler overlapping subproblems, then efficiently combining the solutions to the subproblems. Assume the string x is of length m, and y is of length n. Let $x[i]$ denote the string consisting of the first i characters of x (if $i = 0$, $x[i]$ is the empty string). We construct a $(m + 1) \times (n + 1)$ table E such that $E_{ij} = \mathrm{ED}(x[i], y[j])$ (where $0 \leq i \leq m$ and $0 \leq j \leq n$). This can be done using the observation (which should be checked!) that

$$E_{ij} = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\{E_{i-1,j} + 1, E_{i,j-1} + 1, E_{i-1,j-1} + [x_i \neq y_j]\} & \text{otherwise.} \end{cases}$$

Here we define $[x_i \neq y_j] = 1$ if $x_i \neq y_j$ and $[x_i \neq y_j] = 0$ if $x_i = y_j$. Thus we can first fill in the first row and column of E (which do not depend on x and y), then fill the remaining entries of E row by row, starting at the top left. $\mathrm{ED}(x, y)$ is then the bottom-right entry in E. As there are $(m+1)(n+1)$ entries in E, and each requires $O(1)$ time to compute, the time required to compute $\mathrm{ED}(x, y)$ is $O(mn)$. This process is illustrated in Figure 1.1. □

		a	b	b	a
	0	1	2	3	4
b	1	1	1	2	3
a	2	1	2	2	2
c	3	2	2	3	3
a	4	3	3	3	3

Figure 1.1: A completed table demonstrating that $\mathrm{ED}(baca, abba) = 3$.

Finding a Maximum Matching is in P

This example problem is graph-theoretic in nature: finding a maximum matching in a bipartite graph. A *matching* M in an undirected graph G is a subset of the edges of G such that no pair of edges has a vertex in common. M is said to be a *maximum* matching if it is as large as possible, and *perfect* if every vertex in G is included in an edge of M. Finally, G is said to be bipartite if its vertices can be partitioned into sets L and R such that every edge in G connects a vertex in L to a vertex in R. See Figure 1.2 for an illustration.

If we let the sets L and R correspond to "workers" and "jobs", and an edge correspond to "this worker can do that job", G has a perfect matching if and only if every job can be assigned to a worker. A simple criterion is known for whether such a perfect matching exists.

Theorem 15 (Hall's marriage theorem)**.** *Let $G = (V, E)$ be a bipartite undirected graph with bipartition $V = L \cup R$, and $|L| = |R|$. Then G has a perfect matching if and only if, for all $X \subseteq L$, $|X| \leq |\{(x, y) : x \in X, (x, y) \in E\}|$.*

Observe that Hall's marriage theorem does not imply an efficient algorithm to determine whether G has a perfect matching. If $|L| = |R| = n$, to verify the conditions of the theorem could require testing 2^n different subsets of L. However, a different approach does lead to an efficient algorithm; in fact, an efficient algorithm for the more general task of *finding* a maximum matching.

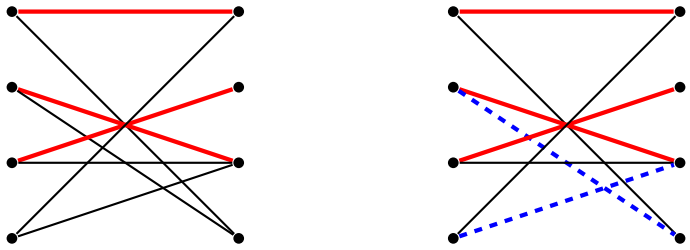


Figure 1.2: A bipartite graph with a (non-maximum) matching, and an augmenting path with respect to that matching, which results in a perfect matching.

The approach we will use is known as the augmenting path algorithm. Let $M \subseteq E$ be a matching in a bipartite graph $G = (V, E)$. We say that $v \in V$ is unmatched if v is not included in M. An *augmenting path* for M is a sequence of edges that starts and ends with an unmatched vertex and alternates between edges of $E \backslash M$ and edges of M. The algorithm proceeds as follows.

1. Set M to $\emptyset$.
2. While there is an augmenting path P for M, replace M with $M \Delta P$.
3. If there is no augmenting path, return M.

From the definition of an augmenting path, we see that replacing M with $M \Delta P$ increases the size of M by 1; a little thought shows that the resulting set is still a matching. Augmenting paths have the following additional nice property.

Lemma 16. *Let M be a matching. If M is not maximum, there exists an augmenting path for M.*

Proof. Assume that M is not maximum, let N be a maximum matching, and consider the graph with edges $X = M \Delta N$. This graph is of degree at most 2, so consists of a collection of disconnected paths and cycles, where a cycle is a path whose initial vertex is the same as its final vertex, i.e. a path of the form $v_1 \to v_2 \to v_3 \to \ldots v_1$. Each path and cycle alternates between edges from M and edges from N. As $|N| > |M|$, X contains more edges from N than from M. Each cycle in X has the same number of edges from each, so there must exist a path with exactly one more edge from N than from M. This is an augmenting path as desired. □

This lemma implies that the augmenting path algorithm is correct (i.e. will always output a maximum matching of M). The remaining question is how to find such an augmenting path efficiently, which can be achieved as follows for bipartite graphs G. Form a directed graph from G by directing edges from L to R if they do not belong to M, and from R to L if they do, and call this directed graph G'. Then any path in G' from an unmatched node in L to an unmatched

node in R corresponds to an augmenting path in G with respect to M. In order to find such a path, we add a new node s to G', and add an arc from s to every unmatched node in L. We then apply the algorithm for PATH, starting at s and terminating when we find an unmatched node in R. The path which is output gives our desired augmenting path. The whole algorithm runs in time poly(n).

Self Test Question:
Given two strings $a, b \in \Sigma^*$, a common subsequence of a and b is a sequence of symbols which is a subsequence of both a and b. For example, *abara* is a common subsequence of *abracadabra* and *barbarian*. Give an algorithm which outputs the maximal length of a common subsequence of two strings in polynomial time.

Solution:
The algorithm uses dynamic programming to create a table L whose (i, j)'th entry $L_{i,j}$ is the length of the longest common subsequence between the first i characters of a and the first j characters of b, where $i \geq 0$, $j \geq 0$. Now observe that

$$L_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{if } a_i \neq b_j \\ L_{i-1,j-1} + 1 & \text{if } a_i = b_j. \end{cases}$$

That is, the longest common subsequence (LCS) of any string and the empty string is empty; the LCS of any pair of strings x and y that differ on the last letter cannot include that letter, so is the maximum of the LCSes of the substrings formed by deleting the last letter from each of x and y; on the other hand, if the strings have the same last letter, the LCS must include that letter. L can now be filled in from the top left in constant time per letter, in the same way as in the algorithm for edit distance. If a is length n and b is length m, the total time required is $O(nm)$.

1.7 Advanced Technical Notes

Self Test Question:
Let DNF-SAT be the variant of SAT where the input is required to be a boolean formula in DNF, rather than CNF. Prove that DNF-SAT $\in$ P, and consider the following claim: As CNF formulae can be converted into DNF formulae using De Morgan's laws, P = NP. Is this claim correct?

Solution:
A boolean formula ϕ in DNF is satisfiable if and only if any one of its clauses is satisfiable. Therefore, if ϕ contains at least one clause, and that clause is internally consistent (i.e. does not contain $x_i \wedge \neg x_i$ for some i), then ϕ is satisfiable; otherwise, it is not. This can be checked in polynomial time. The claim is not correct because applying De Morgan's laws to a CNF formula ϕ gives a DNF formula ϕ' with an $\neg$ at the front. Satisfiability of ϕ' does not imply satisfiability of ϕ.

Self Test Question:
Prove that HALT is NP-hard. Is it NP-complete?

Solution:
We reduce SAT to HALT. Consider the following algorithm: given a boolean formula ϕ, try every possible assignment x to ϕ. If a satisfying assignment is found, then halt; otherwise, loop forever. Determining whether ϕ is satisfiable thus reduces to HALT. But HALT cannot be NP-complete, as it is undecidable, while every problem in NP is decidable.

Self Test Question:
Prove that, for any $\epsilon > 0$, there exists an NP-complete language which can be decided in time $O(2^{n^\epsilon})$. ("There exist almost arbitrarily easy NP-complete problems.")

Solution:
Take a SAT instance ϕ of size n and pad the input to size n^k with zeroes, for arbitrary constant $k = 1/\epsilon$. Then the zeroes can be ignored, and trying each possible assignment to ϕ, satisfiability can be decided in time $O(2^{n^\epsilon})$.

Lecture 2

Nondeterministic Turing Machines and the Cook-Levin Theorem

2.1 Nondeterministic Turing machines

Nondeterministic Turing machines (NDTMs) are (nonphysical and unrealistic!) generalisations of the Turing machine model. Instead of having one transition function δ, an NDTM may have several functions $\delta_1, \ldots, \delta_K$. A computational path is a sequence of choices of transition functions (i.e. integers between 1 and K). We think of an NDTM M as making all of these transitions in parallel, leading to many different potential computational paths. It is kind of like a many universe model of reality, where every decision you make spawns another possible universe[1]. See Figure 2.1 for an illustration of this. NDTMs also have a special ACCEPT state.

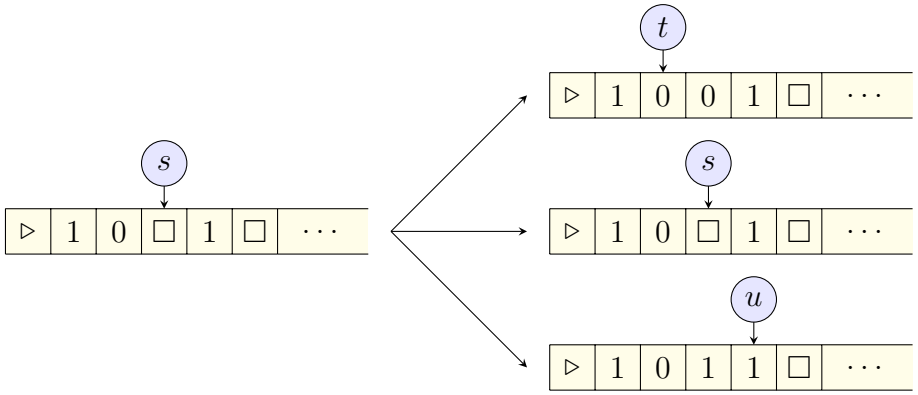


Figure 2.1: An NDTM can take multiple paths "in parallel".

We say that an NDTM M decides a language $\mathcal{L} \subseteq \{0,1\}^*$ if:

[1]Read its nuts! This way of think of NP is what I meant by "old fashioned in the introduction. Nowadays people think in terms of witnesses etc, but we still need this notion to be able to give the proof of Cook-Levin

- All M's computational paths reach either the state HALT or the state ACCEPT;
- For all $x \in \mathcal{L}$, on input x at least one path reaches the ACCEPT state;
- For all $x \notin \mathcal{L}$, on input x no path reaches the ACCEPT state.

We say that M runs in time $T(n)$ if, for every input $x \in \{0,1\}^*$ and every sequence of nondeterministic choices, M reaches either the state HALT or the state ACCEPT in at most $T(|x|)$ steps. We can now define the class NTIME as follows, by analogy with DTIME:

Definition 6. *A language $\mathcal{L} \subseteq \{0,1\}^*$ is said to be in $\mathsf{NTIME}(T(n))$ if there is an NDTM that runs in time $cT(n)$ for some constant $c > 0$ and decides $\mathcal{L}$.*

Unlike standard Turing machines, we do not expect to be able to actually implement NDTMs. However, they will nevertheless be a very useful conceptual tool. We first observe that there is no loss of generality in assuming that $K = 2$, i.e. that M has only two possible transition rules. Indeed, for any $K \geq 2$ one can associate each transition rule δ with a string of $\lceil \log_2 K \rceil$ bits and create new rules δ'_0, δ'_1 which, over $\lceil \log_2 K \rceil$ steps, select the bits of δ (storing each bit in a different head state).

Theorem 17. $\mathsf{NP} = \bigcup_{c \geq 1} \mathsf{NTIME}(n^c)$.

Proof. First, suppose $\mathcal{L} \in \mathsf{NP}$. Then there exists a polynomial-time verifier V such that, for all $x \in \mathcal{L}$, there is a certificate w of size $\text{poly}(|x|)$ such that V accepts on input (x, w); and for all $x \notin \mathcal{L}$, no certificate w exists such that V accepts on input (x, w). To decide $\mathcal{L}$, on input x our NDTM M simply guesses w nondeterministically and runs V on (x, w). To make this idea of "guessing" concrete, we imagine that M has two special transition functions δ_0, δ_1, which correspond to writing a 0 or a 1 to the tape, allowing M to guess each bit of w in turn and write it to the tape.

Second, suppose M is a polynomial-time NDTM that decides $\mathcal{L}$. Then, for each $x \in \mathcal{L}$, there is a computational path of length $\text{poly}(|x|)$ leading to the ACCEPT state, but for all $x \notin \mathcal{L}$, there is no such path. This serves as a certificate: the verifier takes as input (x, p), where p identifies a computational path, and just simulates the path p on input x. □

One can similarly define the nondeterministic analogue of EXP, $\mathsf{NEXP} = \bigcup_{c \geq 1} \mathsf{NTIME}(2^{n^c})$.

2.2 Proof of the Cook-Levin Theorem

The Cook-Levin Theorem was proven by Stephen Cook in 1971, and independently by Leonid Levin in 1973, on the other side of the Iron Curtain. Every good textbook on computational complexity includes a proof of this fundamental theorem. The proof here is largely based on that in Sipser (chapter 7).

We already know that SAT $\in$ NP. To show that SAT is in fact NP-complete, we need to show that every language in NP reduces to SAT. To be precise, we will show that, for any language $\mathcal{L} \subseteq \{0,1\}^*$ such that $\mathcal{L} \in \mathsf{NP}$, given $x \in \{0,1\}^n$, we can construct (in time

poly(n)) a CNF formula ϕ such that ϕ is satisfiable if and only if $x \in \mathcal{L}$. As $\mathcal{L} \in \mathsf{NP}$, there is a polynomial-time NDTM M that decides $\mathcal{L}$. Given a description of M, we will encode this as a formula ϕ, which will evaluate to true if and only if M has an accepting path on x.

$\triangleright$	START	x_1	x_2	$\ldots$	x_n	$\square$	$\ldots$	$\square$
$\triangleright$	0	s	x_2	$\ldots$	x_n	$\square$	$\ldots$	$\square$
			$\vdots$					
$\triangleright$	0	1	ACCEPT	$\ldots$	1	0	$\ldots$	$\square$

Figure 2.2: A tableau describing a particular computational path.

As M is a polynomial-time NDTM, there is a constant c such that M runs in time at most $T = n^c$ on input x. We associate a $(T+1) \times (T+2)$ tableau with each computational path of M, where row t contains the configuration of M at step t of the path; see Figure 2.2 for an illustration. Each row thus consists of a triple (ℓ, q, r), where q is a head state and ℓ and r are strings of tape symbols. Then our CNF formula is made up of subformulae,

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

where

- ϕ_{cell} evaluates to true if all squares in the tableau are uniquely filled;
- ϕ_{start} evaluates to true if the first row is the correct starting configuration;
- ϕ_{move} evaluates to true if the tableau is a valid computational path of M (according to its transition rules);
- ϕ_{accept} evaluates to true if the tableau contains an accepting state.

It should be intuitively clear that there is a satisfying assignment to ϕ if and only if there is a valid tableau (i.e. computational path that accepts x). Formally, introduce a set of boolean variables $c_{i,j,s}$, each of which evaluates to true if cell (i,j) contains symbol s. Then

$$\begin{aligned}
\phi_{\text{accept}} &= \bigvee_{i,j} c_{i,j,\text{ACCEPT}}; \\
\phi_{\text{start}} &= c_{1,1,\triangleright} \wedge c_{1,2,\text{START}} \wedge c_{1,3,x_1} \wedge c_{1,4,x_2} \wedge \cdots \wedge c_{1,n+2,x_n} \wedge c_{1,n+3,\square} \wedge \cdots \wedge c_{1,T+2,\square}; \\
\phi_{\text{cell}} &= \bigwedge_{i,j} \left(\left(\bigvee_{s} c_{i,j,s} \right) \bigwedge_{t \neq u} (\neg c_{i,j,t} \vee \neg c_{i,j,u}) \right).
\end{aligned}$$

The last of these encodes the constraint that every cell has a value, and no cell has two values, and can easily be written in CNF. For the final constraint ϕ_{move}, we need to encode the validity of a computational path. The key insight is that this can be done using the *locality* of Turing machines. Consider a 2×3 "window" (submatrix) in a tableau. If S is the

number of possible symbols used to write a configuration, there are only poly(S) possible windows – a fixed, finite number. However, some of these are disallowed ("illegal") because they correspond to transitions which cannot take place. For example, the head cannot move two positions in one step, so if s is a head state the window

s	0	0
0	0	s

is illegal. In general, whether a given window is legal or not will depend on the transition functions of M. The constraint that a window w is legal can be written as

$$\phi_w = \bigvee_{\text{legal windows } v} [w_{ij} = v_{ij} \text{ for } i \in \{1,2\}, j \in \{1,2,3\}].$$

Note that this constraint is a boolean formula with a fixed, finite number of terms, which by Theorem 11 can be written in CNF.

We claim that, if all windows in a tableau are legal, then each row in the tableau corresponds to a configuration which legally follows the previous one, i.e. the tableau as a whole corresponds to a valid computational path. Intuitively, this is because the head can only move one step at a time, so every possible transition will occur within some window. The (somewhat more formal) proof is by induction. For the base case, ϕ_{start} ensures that the first row is valid. We now show that, if row r is a valid configuration, then row $r+1$ is also a valid configuration. Consider any window onto rows r and $r+1$. If the window does not contain a head symbol in its top row, in order for the window to be legal the middle symbol in its bottom row must be equal to the middle symbol in its top row, so all tape cells which are not adjacent to the head are preserved[2]. On the other hand, as row r is a valid configuration, there must exist a window containing a head symbol in the middle of its top row. This window will only be legal if its bottom row corresponds to a valid transition (and in particular also contains a head symbol). And as the head can only move at most one position per step, a window whose first row does not contain a head symbol cannot contain a head symbol in the middle of its bottom row, so row $r+1$ must contain exactly one head symbol. Thus row $r+1$ is a valid configuration.

We can therefore write

$$\phi_{\text{move}} = \bigwedge_{\text{windows } w} \phi_w.$$

Thus, combining the four subformulae, ϕ being satisfiable is equivalent to M accepting x. Finally, observe that ϕ is of size poly(T) = poly(n), and can be produced in time poly(n) from a description of M. This completes the proof.

[2]The alert reader may wonder about the special cases of the first and last columns of the tableau. These can be dealt with either by having special constraints on whether windows involving these columns are legal, or introducing special "border" symbols to identify the start and end of each row.

2.3 More NP-complete problems

It turns out that many, many interesting problems are known to be NP-complete. Garey and Johnson alone list over 300! The Cook-Levin theorem allows us to prove a given problem to be NP-complete in a much more straightforward manner than was the case for SAT. Indeed, if we can show, for some language $\mathcal{L} \in \mathsf{NP}$, that SAT $\leq_{\mathsf{P}} \mathcal{L}$, then $\mathcal{L}$ is NP-complete. Here are a few examples of other problems which are known to be NP-complete.

- Clique: given an undirected graph G and an integer k, does G contain a clique on k vertices, i.e. a k-subset of the vertices such that every pair of vertices in the subset are connected by an edge?
- Hamiltonian Path: given a directed graph G, does it contain a path visiting each node exactly once?
- Subset Sum: given a sequence S of n integers and a "target" t, is there a subsequence of S that sums to t?
- Quadratic Diophantine Equation: given positive integers a, b and c, are there positive integers x and y such that $ax^2 + by = c$?
- Shortest Common Superstring: given a finite set of strings $S \subset \{0,1\}^*$ and an integer k, is there a string $s \in \{0,1\}^*$ such that $|s| \leq k$ and each $x \in S$ is a substring of s?
- 3-Colouring: given a graph G, can the vertices each be assigned one of three colours, such that adjacent vertices receive different colours?

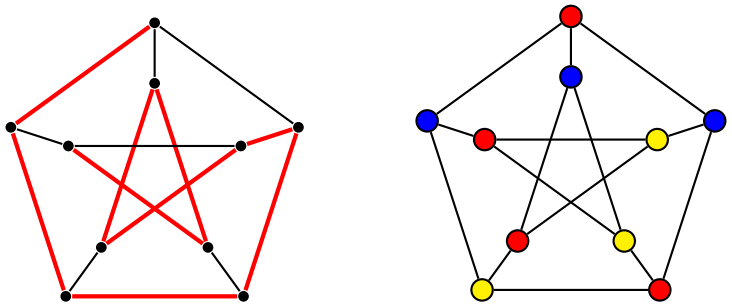


Figure 2.3: Examples of a Hamiltonian path in an undirected graph and a 3-colouring.

We will discuss many of these problems in the Supplementary Notes section, but we end with proving the theorem

Theorem 18. Clique *is* NP*-complete, assuming* 3-SAT *is* NP*-complete.*

In the Supplementary Notes we will also prove 3-SAT is NP-complete. We present the following proof as an example of how one establishes a new problem is NP-complete (by reducing a problem in NP to a problem already known to be NP-complete).

Proof. CLIQUE is clearly in NP, as given a claimed clique on m vertices, it can be verified in time $O(m^2)$. To show that CLIQUE is NP-complete, we reduce 3-SAT to CLIQUE as follows. Given a 3-CNF formula with m clauses, we create an undirected graph G with at most $3m$ vertices. We associate a set of at most three vertices with each clause C, labelled by C and the literals in C. We then connect every pair of vertices in G, with the exception of vertices in the same clause and vertices labelled with a variable and its negation (i.e. $(x_i, \neg x_i)$ for some i). See Figure 2.4 for an illustration.

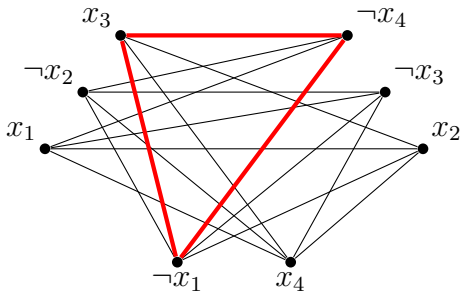


Figure 2.4: Reducing satisfiability of the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$ to finding a 3-clique (one such clique is highlighted).

We now show that G has an m-clique if and only if ϕ is satisfiable.

- (Satisfiable $\Rightarrow$ m-clique): Let w be a satisfying assignment to ϕ. We form a set S of variables by picking a variable x_i from each clause C such that $x_i = w_i$ and x_i satisfies C (this is always possible as w satisfies every clause of ϕ). Then there is an edge from every element of S to every other element, because each x_i is consistent with the fixed variable w_i, so we never have an edge $(x_i, \neg x_i)$.
- (m-clique $\Rightarrow$ satisfiable): Any m-clique must include exactly one vertex from each clause, as there are no edges between vertices in the same clause. Also, the labels for the vertices must be consistent, as there are no edges between inconsistent labellings. Thus taking the labelling of each vertex in the clique (and assigning any other variables arbitrarily) gives a satisfying assignment for ϕ.

As G can clearly be constructed from ϕ in polynomial time, this completes the proof. □

2.4 Supplementary Notes

The theory of NP-completeness took off in 1972 when Karp proved that the decision versions of many practically important optimisation problems are NP-complete; literally thousands of other

problems have since been proven NP-complete, and this terminology has now entered the standard scientific vocabulary. Garey and Johnson is the standard reference for NP-completeness results (albeit now somewhat outdated).

Could it be the case that *all* problems in NP are either in P or NP-complete? Assuming that $\mathsf{P} \neq \mathsf{NP}$, it has been shown that this cannot be true: there must exist problems in NP which are neither in P nor NP-complete ("Ladner's theorem", see Papadimitriou chapter 14). However, very few natural candidate problems are known which are thought to belong to $\mathsf{NP} \backslash \mathsf{P}$. One example is Integer Factorisation. Another is Graph Isomorphism: given the description of two undirected graphs G and H, is there a permutation π of the vertices of G such that $\pi(G) = H$?

3-SAT

The first NP-completeness proof we will give is for a restricted variant of SAT, which turns out to be very useful for other such proofs. We say that ϕ is a k-CNF formula if it is a boolean formula in CNF with at most k variables per clause. Let k-SAT denote the special case of the SAT problem where the input is restricted to k-CNF formulae. For example,

$$\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_5 \vee x_7) \wedge (x_3)$$

is a valid 3-SAT instance.

Theorem 19. *3-SAT is* NP*-complete.*

Proof. As 3-SAT is a restriction of SAT, it is clearly in NP. To show that it is NP-hard, we reduce SAT to 3-SAT. Given as input a boolean formula ϕ in CNF, we will obtain a new formula ϕ' by replacing each clause which contains $k \geq 4$ literals $(\ell_1, \ldots, \ell_k)$ with $k-2$ clauses which contain 3 literals, while preserving satisfiability of ϕ. Consider the formula

$$\phi_{\ell_1,\ldots,\ell_k} = (\ell_1 \vee \ell_2 \vee u_1) \wedge (\ell_3 \vee \neg u_1 \vee u_2) \wedge \cdots \wedge (\ell_{k-2} \vee \neg u_{k-4} \vee u_{k-3}) \wedge (\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}),$$

where we have introduced $k-3$ new variables $u_1, \ldots, u_{k-3}$. If any of the original literals ℓ_i evaluates to true, then the whole formula is satisfiable (taking $u_j = 1$ for u_j up to the clause in which ℓ_i is encountered, and $u_j = 0$ thereafter). On the other hand, if none of the ℓ_i evaluate to true, $\phi_{\ell_1,\ldots,\ell_k}$ is equivalent to

$$u_1 \wedge (\neg u_1 \vee u_2) \wedge \cdots \wedge (\neg u_{k-4} \vee u_{k-3}) \wedge (\neg u_{k-3}),$$

and a moment's thought shows that this is not satisfiable. Thus the resulting overall formula ϕ' obtained by replacing all clauses containing more than 3 literals in this way is satisfiable if and only if ϕ was satisfiable, so SAT $\leq_{\mathsf{P}}$ 3-SAT. □

2-SAT

Can we improve this result to show that 2-SAT is NP-complete? Perhaps surprisingly, we have the following result suggesting that we cannot.

Theorem 20. *2-SAT* $\in$ P*.*

Proof. We will reduce 2-SAT to Path. Given an n-variable 2-SAT formula ϕ with exactly two distinct variables per clause (clauses containing only one variable can be dealt with trivially), we construct a directed graph G as follows.

- G has $2n$ nodes labelled by literals (i.e. variables and their negations) appearing in ϕ.
- For each clause $(\ell_1 \vee \ell_2)$, G has arcs $\neg\ell_1 \to \ell_2$ and $\neg\ell_2 \to \ell_1$.

These arcs are intended to symbolise logical implication. If we have a clause $(x_1 \vee \neg x_2)$, for example, we can think of it as capturing the implication $x_2 \Rightarrow x_1$ ("if $x_2 = 1$, then $x_1 = 1$") or equivalently the contrapositive $\neg x_1 \Rightarrow \neg x_2$. ϕ is satisfiable if and only if all of these implications are consistent.

We now show that ϕ is unsatisfiable if and only if there is a variable x_i such that there is a path from x_i to $\neg x_i$ and from $\neg x_i$ to x_i.

- ($\Leftarrow$): Suppose for a contradiction that there is a path from x_i to $\neg x_i$ and from $\neg x_i$ to x_i, and yet ϕ is satisfiable. Let T be a satisfying assignment to the variables. If $T(x_i) = 1$, then $T(\neg x_i) = 0$, so there must be an arc $\ell_1 \to \ell_2$ on the path from x_i to $\neg x_i$ such that $T(\ell_1) = 1$, $T(\ell_2) = 0$. This means that $(\neg\ell_1 \vee \ell_2)$ is a clause of ϕ, but is not satisfied by T. The case $T(x_i) = 0$ is similar.
- ($\Rightarrow$): We consider the following method for finding a satisfying assignment to ϕ by assigning truth values to each node of G. Repeatedly, for each node ℓ which has not yet been assigned a value, and such that there is no path from ℓ to $\neg\ell$, we assign 1 to ℓ and each node reachable from ℓ; we also assign 0 to $\neg\ell$ and the negations of the nodes reachable from ℓ (i.e. the nodes from which $\neg\ell$ is reachable).

  We first check that this process makes sense (i.e. assigns consistent values to the literals). If there were a path from ℓ to two nodes m and $\neg m$, then by symmetry of G there would be paths from both m and $\neg m$ to $\neg\ell$, so there would be a path from ℓ to $\neg\ell$, contradicting the hypothesis. So, in each step, truth values are assigned consistently. Further, if there is a path from ℓ to a node m previously assigned 0 (say), as m is reachable from ℓ, ℓ must have also previously been assigned 0.

  As we have assumed that for each variable x_i, there is either no path from x_i to $\neg x_i$, or no path from $\neg x_i$ to x_i, all x_i will be assigned a value. As the assignment is produced such that whenever $\ell \to m$, either m is assigned 1 or ℓ is assigned 0, the assignment satisfies ϕ.

We can therefore apply the polynomial-time algorithm for PATH given in Theorem 10 to each pair $(x_i, \neg x_i)$ to solve 2-SAT in polynomial time. □

The Subset Sum Problem

The next problem we consider is a basic question from arithmetic, the SUBSET SUM problem[3]: given a sequence S of n integers and a "target" t, is there a subsequence of S that sums to t?

Theorem 21. SUBSET SUM *is* NP*-complete.*

Proof. It is obvious that SUBSET SUM is in NP (the certificate is simply the subsequence of S that sums to the target). To prove that it is NP-complete, we reduce SAT to SUBSET SUM. Imagine we are given an n-variable formula ϕ in CNF with clauses $C_1, \ldots, C_m$ where each clause uses at most ℓ variables. We create some $(n+m)$-digit numbers (in base $\ell+1$) that sum to a particular target t if and only if ϕ is satisfiable. These numbers have one column corresponding to each variable, and one to each clause.

[3]The version of the problem we describe here would technically be better known as SUBSEQUENCE SUM.

- For each variable v, create two integers n_{vt} and n_{vf} which are both 1 in the column corresponding to v. n_{vt} also has a 1 digit in the columns corresponding to clauses that are satisfied by v being true, and 0 digits elsewhere. n_{vf} also has a 1 digit in the columns corresponding to clauses that are satisfied by v being false, and 0 digits elsewhere.
- For each clause C_i containing ℓ_i variables, create $\ell_i - 1$ numbers that are 1 in the column corresponding to C_i, and 0 elsewhere.
- The target t has 1 digits in all the columns corresponding to variables, and a digit equal to ℓ_i in the column corresponding to C_i.

Now, if there is a subsequence of integers summing to t, this must use exactly one of each of the pairs $\{n_{vt}, v_{vf}\}$ (corresponding to each variable being either true or false) to make the first n columns correct. Also, each clause of ϕ must have been satisfied by at least one variable, or the last m columns cannot be correct. Therefore, there is a satisfying assignment to ϕ if and only if there is a subsequence of the numbers that sums to t. □

Hamiltonian Path

The strategy used in the above proof of associating a subgraph with each fragment of the original formula, then connecting the subgraphs appropriately, is known as a *gadget construction* and is frequently a useful idea.

Theorem 22. Hamiltonian Path *is* NP*-complete.*

Proof. Hamiltonian Path is in NP because a claimed path visiting every vertex once can be checked in polynomial time. To show that the problem is NP-complete, we reduce SAT to Hamiltonian Path. Given a formula ϕ with n variables and m clauses, we will produce a (directed) graph G in polynomial time such that G has a Hamiltonian path if and only if ϕ is satisfiable.

G will have m nodes v_{C_i} corresponding to the clauses $C_1, \ldots, C_m$ of ϕ; n "chains" of $2m + 1$ nodes each corresponding to the variables $x_1, \ldots, x_n$; and finally two special nodes v_{in}, v_{out}. A chain is a sequence of nodes v_i such that each pair v_i, v_{i+1} are connected by arcs in both directions.

We also have arcs from v_{in} to the start and end of the first chain, and from the start and end of the last chain to v_{out}; and from the start and end of the j'th chain to the start and end of the $j + 1$'th chain, for each j. Finally, we have arcs corresponding to the clauses of ϕ, as follows. If clause C contains the (unnegated) variable x_i, then we take two neighbouring nodes v_j, v_{j+1} in the i'th chain, and put an arc from v_j to v_C, and from v_C to v_{j+1}. If C contains the negation $\neg x_i$, we instead put an arc from v_C to v_j, and from v_{j+1} to v_C. We never use the same node or arc for two different clauses (as each chain contains $2m + 1$ nodes, we can always achieve this). This whole construction is illustrated in Figure 2.4.

We now need to show that G has a Hamiltonian path if and only if ϕ is satisfiable.

- (Satisfiable ⇒ Hamiltonian path): Assume ϕ has satisfying assignment x. The path will start at v_{in} and traverse each chain in turn, finishing at v_{out}. The path traverses the i'th chain from left to right if $x_i = 1$, and from right to left if $x_i = 0$. This is not quite a Hamiltonian path as it does not visit the "clause" vertices. But, as x is a satisfying assignment to ϕ, for each clause C there must exist at least one chain such that the path can visit C en route from the start to end of that chain (i.e. the path is going in the right direction along that chain).

- (Hamiltonian path ⇒ satisfiable): First observe that any Hamiltonian path P must begin in v_{in} and end in v_{out}. We now claim that P needs to traverse each chain in order (each either from left to right, or right to left). If there were no clause nodes, this would be obvious; however, it might be the case that P goes from node v_i on the j'th chain to a clause node C, then goes from C to a different chain k. Nevertheless, this can never give a Hamiltonian path: when P reaches the neighbouring node v_{i+1} (or v_{i-1}) it will get stuck, as there are no arcs leaving this neighbour to nodes which have not already been visited. Hence forming a bit-string x by taking $x_i = 1$ if the i'th chain is traversed from left to right, and $x_i = 0$ if the i'th chain is traversed from right to left, gives a satisfying assignment to ϕ.

□

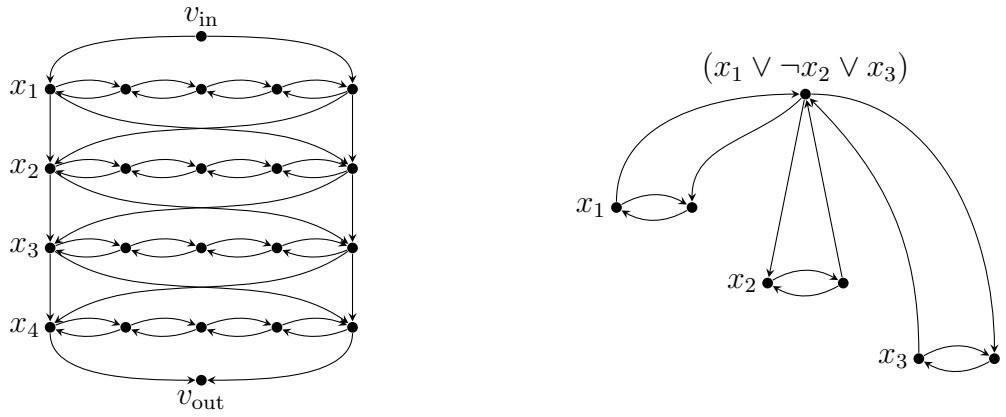


Figure 2.5: Parts of the reduction from determining satisfiability of $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$ to finding a Hamiltonian path. The two pictures illustrate the chains without the clause nodes, and one clause node.

It may be already apparent that, although NP-completeness proofs can usually be readily checked, devising a suitable reduction to prove NP-completeness can require some ingenuity. This is perhaps appropriate given the definition of NP.

Self Test Question:
Prove that the following problem is NP-complete:

0-1 Integer Programming. Given a list of linear inequalities over n variables with integer coefficients, is there an assignment of 0's and 1's to the variables that satisfies all the inequalities?
For example, the inequalities

$$3x_1 - x_2 + x_3 \geq 1 \text{ and } -x_1 + 2x_3 \leq 2$$

are satisfied by setting $x_1 = 1$, $x_2 = 0$, $x_3 = 1$.

Solution:
We reduce 3-SAT to 0-1 Integer Programming. Given a boolean formula ϕ, create an instance of 0-1 Integer Programming as follows: for each clause $\ell_1 \vee \ell_2 \vee \ell_3$, introduce an inequality $\ell'_1 + \ell'_2 + \ell'_3 \geq 1$, where $\ell'_i = y_i$ if ℓ_i appears unnegated in the clause, and $\ell'_i = 1 - y_i$ if ℓ_1 appears negated. For example, the clause $x_1 \vee \neg x_2 \vee x_3$ gives the inequality $y_1 + 1 - y_2 + y_3 \geq 1$. Each such inequality is satisfied if and only if the original clause is satisfied.

Self Test Question:
Prove that the following problem is NP-complete:

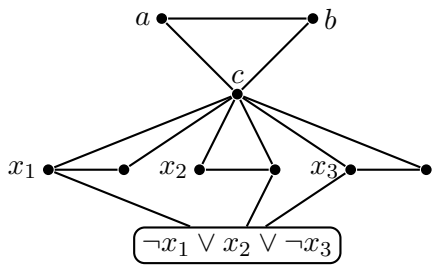
Hamiltonian Cycle. Given a directed graph G, is there a cycle which includes each vertex of G exactly once?

Solution:
We reduce Hamiltonian Path to Hamiltonian Cycle. Given G, create G' by introducing a new node v' which is connected to every node in G in both directions. Then any Hamiltonian path $v \to \cdots \to w$ in G gives a Hamiltonian cycle $v' \to v \to \cdots \to w \to v'$ in G'; similarly, any Hamiltonian cycle in G' gives a Hamiltonian path in G by removing v' from the cycle.

Self Test Question:
Prove that the following problem is NP-complete:

3-colouring. Given an undirected graph G, does there exist an assignment of at most 3 colours to the vertices of G, such that adjacent vertices are assigned different colours?

Solution:
(Sketch.) We reduce 3-SAT to 3-COLOURING. Given a 3-SAT instance ϕ, the idea is to create gadgets corresponding to variables and clauses to enforce the fact that each clause of ϕ must be satisfied. The following picture shows a fragment of this construction corresponding to three variables and one clause.



Each variable x_i corresponds to a pair of vertices f_i, t_i, which are connected to each other and to vertex c. Assume (renaming colours if necessary) that vertices a, b, c are assigned colours 0, 1, 2 respectively. Then, in each gadget corresponding to a variable x_i, t_i and f_i must be assigned colours 0 and 1, or 1 and 0. The first of these encodes $x_i = 0$, the second $x_i = 1$. We now describe the clause gadgets.
Each such gadget is designed to be 3-colourable if and only if at least one of the variables in that clause satisfies that clause. To achieve this, we design an OR gadget: a gadget which is only colourable if at least one of two "incoming" vertices v, w is not coloured with 0. The gadget consists of a triangle where one of the vertices is connected to v, one to w and one (the "output" vertex) to the fixed vertex a. It is easy to convince oneself that this gadget can be 3-coloured if and only if at least one of the incoming vertices is not coloured with 0. By connecting two OR gadgets, we get a gadget which can be 3-coloured if and only if at least one of the three incoming vertices is not coloured with 0. We then use an OR gadget for each clause; for each variable x_i that appears in the clause, we attach the gadget to vertex t_i if x_i appears unnegated, and f_i if x_i appears negated. The resulting graph is 3-colourable if and only if ϕ is satisfiable.
For an illustration of a (slightly different version of) this procedure, see `http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/21-nphard.pdf`.

Self Test Question:
Consider the following variants of the SAT problem. In each case, either show that the problem is in P or that it is NP-complete, as appropriate.

1. The special case of 3-SAT where each variable appears at most three times in the input formula.

2. The special case of 3-SAT where each variable appears at most twice in the formula.

3. The variant of SAT where a clause is *unsatisfied* if and only if the literals in the clause either all evaluate to true, or all evaluate to false. For example, the clause $(x_1, \neg x_2)$ is satisfied by the assignment $x_1 = 1$, $x_2 = 1$ and the assignment $x_1 = 0$, $x_2 = 0$.

4. The variant of 3-SAT where each clause is satisfied if and only if an odd number of literals in the clause evaluate to true. For example, the clause $(x_1, \neg x_2, x_3)$ is satisfied by $x \in \{000, 011, 110, 101\}$.

Solution:

1. NP-complete. Given a variable x_i that appears $k \geq 3$ times, replace each of the occurrences with a new variable y_{ij}. We would like to enforce the constraint $y_{i1} = y_{i2} = \cdots = y_{ik}$. This is equivalent to the chain of implications $y_{i1} \Rightarrow y_{i2} \Rightarrow \cdots \Rightarrow y_{ik} \Rightarrow y_{i1}$, or in other words the sequence of clauses $(\neg y_{i1} \vee y_{i2}) \wedge (\neg y_{i2} \vee y_{i3}) \wedge \cdots \wedge (\neg y_{ik} \vee y_{i1})$. Thus we end up with a formula in which each variable appears at most 3 times (once in the original formula, and twice in the circle of implications).

2. In P. Consider each variable x_i in a formula ϕ. If x_i appears once in ϕ, it can be removed from the formula by replacing it with 0 or 1, as appropriate, without affecting satisfiability of ϕ. If it appears twice unnegated (or negated), the same holds. So we can assume x_i appears twice, once negated, once unnegated. But observe that, for any sets of literals C, D, the pair of clauses $(\neg x_i \vee C)$, $(x_i \vee D)$ are satisfiable if and only if the clause $(C \vee D)$ is satisfiable. Thus we can remove x_i from ϕ and combine the two clauses. Repeating this procedure determines satisfiability of ϕ in polynomial time.

3. NP-complete. Given a boolean formula ϕ, create a new formula ϕ' by introducing a new variable v and adding v to each clause. Imagine ϕ' is satisfiable (under the variant definition), and in a satisfying assignment $v = 0$. Then this is equivalent to "in each clause, at least one literal is equal to 1", and thus gives a satisfying assignment to ϕ. On the other hand, imagine that ϕ' is satisfiable and in a satisfying assignment $v = 1$. Then we can find a satisfying assignment simply by negating each variable in the assignment (by symmetry of solutions).

4. In P. For each clause (ℓ_1, ℓ_2, ℓ_3), the constraint that an odd number of literals evaluates to true is equivalent to $\ell_1 \oplus \ell_2 \oplus \ell_3 = 1$, which is a linear equation over $\mathbb{F}_2$. Thus the whole formula is satisfied if and only if the corresponding system of linear equations has a solution. As solving systems of linear equations is in P (using Gaussian elimination, for example), so is this problem.

Self Test Question:
Is each of the following special cases of NP-complete problems in P or NP-complete?

1. Undirected Hamiltonian Path. Given an undirected graph G, does there exist a path which visits each vertex exactly once?

2. 2-colouring. Given an undirected graph G, does there exist an assignment of at most 2 colours to the vertices of G, such that adjacent vertices are assigned different colours?

3. Clique of size 100. Given an undirected graph G, does it contain a clique of size at most 100?

Solution:

1. NP-complete via reducing Hamiltonian Path to Undirected Hamiltonian Path. Given a directed graph G, create a new undirected graph G' by introducing vertices v_i, v_m, v_o ("input", "middle" and "output") for each node v, with undirected edges $v_i \leftrightarrow v_m \leftrightarrow v_o$. Each arc $v \to v'$ is replaced with an undirected edge $v_o \leftrightarrow v'_i$. If G has a Hamiltonian path $v \to w \to \ldots$, G' has a Hamiltonian path $v_i \to v_m \to v_o \to w_i \to w_m \to w_o \to \ldots$.

   On the other hand, any Hamiltonian path P in G' must include all vertices v_m. Each vertex v_m can only appear either at the start or end of P, or between v_i and v_o. Thus P must take one of the following six forms:

$$v_i \to v_m \to v_o \to w_i \to w_m \to w_o \to \cdots \to z_i \to z_m \to z_o$$
$$v_m \to v_o \to w_i \to w_m \to w_o \to \cdots \to z_i \to z_m \to z_o \to v_i$$
$$v_o \to w_i \to w_m \to w_o \to \cdots \to z_i \to z_m \to z_o \to v_i \to v_m$$
$$v_o \to v_m \to v_i \to w_o \to w_m \to w_i \to \cdots \to z_o \to z_m \to z_i$$
$$v_m \to v_i \to w_o \to w_m \to w_i \to \cdots \to z_o \to z_m \to z_i \to v_o$$
$$v_i \to w_o \to w_m \to w_i \to \cdots \to z_o \to z_m \to z_i \to v_o \to v_m.$$

   In the former three cases, G must have a directed path $v \to w \to \cdots \to z$; in the latter three cases, G must have a directed path $z \to \cdots \to w \to v$.

2. In P. Start by assigning an arbitrary vertex v an arbitrary colour. If G has a 2-colouring, the neighbours of v must have the other colour, so assign them appropriately. Repeat this process until every vertex in the connected component of G containing v has been assigned a colour, or a contradiction is found (i.e. a vertex which has already been assigned one colour has to be assigned another colour). Repeat for each connected component of G.

3. In P. There are $O(n^{100})$ possible sets of vertices of size 100. Each one can be checked for being a clique in polynomial time, so every possible clique can be checked in polynomial time.

2.5 Advanced Technical Notes

We know that $\mathsf{P} \subseteq \mathsf{NP}$. The million-dollar question[4] is whether this inclusion is strict. This question has vast practical significance (many problems which people want to solve in practice turn out to be NP-complete, some of which are given in Section 2.3), but it also has significant theoretical importance, as it essentially asks whether mathematics can be axiomatised in the following concrete sense. Perhaps the first to recognise the connection between the P vs. NP problem and the

[4]Literally: the P vs. NP problem is one of the Clay Mathematics Institute's Millennium Prize problems, and solving it carries a prize of $1m.

foundations of mathematics was Gödel[5] in 1956.

Recall from Section 0.8 that the problem of determining whether a statement about the natural numbers can be proven from the axioms of Peano arithmetic (the Entscheidungsproblem) is undecidable. In practice, we may be satisfied with the following weaker notion of axiomatisation. Given a statement about the natural numbers, does it have a proof (i.e. a derivation of the statement from the axioms) of a reasonable length[6] in Peano arithmetic? One can formalise this as the following decision problem, which we call THEOREM. Given a statement S of length n about the natural numbers, and an integer k, does S have a proof from the axioms of Peano arithmetic of length at most n^k?

Theorem 23. THEOREM *is* NP*-complete.*

Proof. It is clear that THEOREM is in NP: given a claimed proof of S in any "sensible" formal system, we can check it in time polynomial in the length of the proof. To see that it is NP-hard, we reduce 3-SAT to THEOREM using a process known as arithmetisation. Given a 3-CNF formula ϕ, each clause C containing variables (x_i, x_j, x_k) is replaced by the product $(1 - z_i z_j z_k)$, where $z_i = x_i$ if x_i appears negated in C, and $z_i = (1 - x_i)$ otherwise. We then form a polynomial P_ϕ by taking the product of these polynomials over all clauses C. For example,

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \mapsto (1 - (1 - x_1)x_2(1 - x_3))(1 - x_1(1 - x_3)).$$

Now ϕ will have a satisfying assignment x if and only if there exist $x_1, \ldots, x_n \in \{0, 1\}$ such that $P_\phi(x_1, \ldots, x_n) = 1$, which is a simple statement about the natural numbers. □

Thus, if P = NP, we can determine whether statements have short proofs. More is true: if P = NP, we can even *find* such short proofs.

Theorem 24. *If* P = NP*, for every language* $\mathcal{L} \in$ NP *there exists a polynomial-time Turing machine* M *that, on input* $x \in \mathcal{L}$*, outputs a certificate for* x *with respect to* $\mathcal{L}$ *and* M*. That is, there exists a polynomial-time Turing machine* $V_{\mathcal{L},M}$ *such that* $V_{\mathcal{L},M}(M(x)) = 1$ *if and only if* $x \in \mathcal{L}$*.*

Proof. We first show that the above claim holds for SAT, by showing that given access to a polynomial-time algorithm $\mathcal{A}$ for deciding whether some input formula ϕ is satisfiable, we can find a satisfying assignment w (if one exists) in polynomial time. Our algorithm first runs $\mathcal{A}$ to determine satisfiability of ϕ. If it is satisfiable, we consider each of the two formulae ϕ_0, ϕ_1 produced by fixing $x_1 = 0$, $x_1 = 1$, and use $\mathcal{A}$ to test their satisfiability. At least one of these formulae will be satisfiable; whichever one of them is, we fix x_1 appropriately and continue with x_2, etc. We eventually fix all the variables and have determined a satisfying assignment to ϕ.

In order to see that the above strategy works for *any* $\mathcal{L} \in$ NP, observe that the proof of the Cook-Levin theorem maps a certificate for $\mathcal{L}$ (i.e. a computational path of an NDTM) to a satisfying assignment w of a boolean formula in such a way that the certificate can be retrieved from w. □

Thus, if P = NP, there would exist a polynomial-time algorithm for finding short proofs of theorems: informally speaking, mathematical creativity would be automated. It would seem somewhat surprising if this were possible, and hence most effort to date has concentrated on proving the conjecture that P ≠ NP. Unfortunately, there are several substantial barriers known to proving this conjectured separation of complexity classes.

[5] `http://rjlipton.wordpress.com/the-gdel-letter/`

[6] After all, can a claim really be said to be "true" if the proof is too long to check?

2.5.1 The relativisation barrier

We previously used the technique of diagonalisation to prove both the undecidability of the halting problem and the time hierarchy theorem. It is natural to hope that this technique could also be useful to prove $\mathsf{P} \neq \mathsf{NP}$. However, we will now see a significant problem with using this technique to attack the P vs. NP question, which is based on the concept of oracles.

An oracle Turing machine is a standard Turing machine M which is given access to an *oracle* which (via some magical process!) solves the decision problem for some language $\mathcal{L}$, where $\mathcal{L}$ may not be solvable by M without the oracle. Formally, an oracle Turing machine M has three additional states {QUERY, YES, NO}, and an additional read/write *oracle tape*. The operation of M depends on the input x and an additional language $\mathcal{L}$ which is called the oracle for M. If M enters the QUERY state during the computation, in the next step M enters the state YES if the contents q of the oracle tape satisfy $q \in \mathcal{L}$, and enters the state NO if $q \notin \mathcal{L}$. Let $M^{\mathcal{L}}(x)$ denote the output of M on input x with an oracle for $\mathcal{L}$. Note that M can be either a deterministic or nondeterministic Turing machine.

We also define $\mathsf{P}^{\mathcal{L}}$ (resp. $\mathsf{NP}^{\mathcal{L}}$) to be the set of languages which can be decided by a polynomial-time deterministic (resp. nondeterministic) Turing machine with oracle access to $\mathcal{L}$.

Some simple examples to illustrate the concept of oracles are:

- For any $\mathcal{L} \in \mathsf{P}$, $\mathsf{P}^{\mathcal{L}} = \mathsf{P}$. The inclusion $\mathsf{P} \subseteq \mathsf{P}^{\mathcal{L}}$ holds because allowing oracle calls can only increase the set of languages that can be decided, while the inclusion $\mathsf{P}^{\mathcal{L}} \subseteq \mathsf{P}$ holds because a polynomial-time Turing machine can replace each call to an oracle for $\mathcal{L}$ by simply deciding $\mathcal{L}$ directly.

- Let EC be the language $\{(\underline{M}, x, 1^n) : M \text{ outputs 1 on } x \text{ within } 2^n \text{ steps}\}$. We show that $\mathsf{P}^{\mathrm{EC}} = \mathsf{NP}^{\mathrm{EC}} = \mathsf{EXP}$. On the one hand, calling the oracle for EC once on an input of size n^k allows an arbitrary exponential-time computation to be performed, so $\mathsf{EXP} \subseteq \mathsf{P}^{\mathrm{EC}}$. On the other, given exponential time, a deterministic Turing machine can simulate each of the possible calls made to the EC oracle by an NDTM running in polynomial time (there can be exponentially many possible calls, each of which can be simulated in exponential time), so $\mathsf{NP}^{\mathrm{EC}} \subseteq \mathsf{EXP}$. Hence $\mathsf{EXP} \subseteq \mathsf{P}^{\mathrm{EC}} \subseteq \mathsf{NP}^{\mathrm{EC}} \subseteq \mathsf{EXP}$. We say that "relative to EC, $\mathsf{P} = \mathsf{NP}$".

We now show that, relative to oracles, the P vs. NP question has two contradictory resolutions.

Theorem 25 (Baker, Gill and Solovay)**.** *There exist languages $\mathcal{A}$, $\mathcal{B}$ such that $\mathsf{P}^{\mathcal{A}} = \mathsf{NP}^{\mathcal{A}}$, but $\mathsf{P}^{\mathcal{B}} \neq \mathsf{NP}^{\mathcal{B}}$.*

Proof. We have already proven the first part by taking $\mathcal{A}$ to be the previously discussed language EC. The second part is more technical. For any language $\mathcal{B} \subseteq \{0,1\}^*$, let $U_{\mathcal{B}}$ be the language

$$U_{\mathcal{B}} = \{1^n : \mathcal{B} \text{ contains a string of length } n\}.$$

For every $\mathcal{B}$, $U_{\mathcal{B}}$ is clearly in $\mathsf{NP}^{\mathcal{B}}$, as the NP machine can guess a string $x \in \mathcal{B}$ of length n, which can be checked using the oracle for $\mathcal{B}$. We now show that there exists a language $\mathcal{B}$ such that $U_{\mathcal{B}} \notin \mathsf{P}^{\mathcal{B}}$, so $\mathsf{P}^{\mathcal{B}} \neq \mathsf{NP}^{\mathcal{B}}$.

We construct $\mathcal{B}$ via a choosing process using infinitely many stages; each stage will determine whether a finite-sized set of strings is in $\mathcal{B}$. Initially, $\mathcal{B}$ is empty. For each stage i, let M_i be the oracle deterministic Turing machine represented by (the binary expansion of) i. For technical

reasons, we will need to assume that each Turing machine M is represented by infinitely many such strings i; this can easily be achieved by, for example, allowing representations to be padded by arbitrarily many trailing zeroes.

For each i, choose n to be larger than the length of any string in $\mathcal{B}$. We will ensure that $M_i^{\mathcal{B}}$ does not decide $U_{\mathcal{B}}$ in time at most $2^n/10$. To do so, we run M_i on input 1^n for $2^n/10$ steps. Whenever M_i makes oracle queries corresponding to a string which the choosing process has determined to be in $\mathcal{B}$ (or not), the oracle answers consistent with this. When M_i queries a string q for which it has not yet been chosen whether $q \in \mathcal{B}$, the oracle answers that $q \notin \mathcal{B}$, and fixes $q \notin \mathcal{B}$. After M_i has finished its computation, it must either accept or reject 1^n; we will ensure that in either case its answer is incorrect. If M_i accepts 1^n, we declare that all remaining strings of length n are not in $\mathcal{B}$, so $1^n \notin U_{\mathcal{B}}$. On the other hand, if M_i rejects 1^n, we find a string x of length n not queried by M_i (which can be done because M_i only queried at most $2^n/10$ strings) and declare $x \in \mathcal{B}$, so $1^n \in U_{\mathcal{B}}$. As this holds for all i, $U_{\mathcal{B}}$ is not decided by any deterministic oracle Turing machine running in time $2^n/10$. As every polynomial is smaller than $2^n/10$ for large enough n, and every Turing machine M is represented by infinitely many i, if M runs in polynomial time it cannot decide $U_{\mathcal{B}}$. Thus $U_{\mathcal{B}} \notin \mathsf{P}^{\mathcal{B}}$. □

Baker, Gill and Solovay proved their result in 1975; a proof is given in Arora-Barak chapter 3, including a brief discussion of the conceptual connection to independence results in mathematical logic. The significance of Theorem 25 is that it implies that proof techniques which are "blind" to the existence of oracles will not be able to resolve the P vs. NP problem. Indeed, the proofs of the undecidability of the Halting Problem and the Time Hierarchy Theorem which we saw previously rely only on the following two properties:

- The countability of Turing machines (i.e. the existence of an effective representation of these machines by binary strings);
- The ability of one Turing machine to simulate another efficiently (i.e. without significant time overhead).

Oracle Turing machines also satisfy these properties, but we have seen that the P vs. NP question cannot be resolved for such machines. Any resolution of the problem must therefore use some additional property of standard Turing machines.

Later on, following the introduction of concepts concerned with randomness and circuit complexity, we discuss a different roadblock to proving $\mathsf{P} \neq \mathsf{NP}$, called the natural proofs barrier.

Lecture 3

P vs. NP, Decision vs Search and Worst Case vs Average Case

3.1 Complementation of languages and co-NP

Self Test Question:
Show that if $\mathsf{P} = \mathsf{NP}$ then any language $A \in \mathsf{P} \setminus \{\emptyset, \Sigma^*\}$ (that is, any language in P, except the "empty" language containing no words and the "full" language containing every word) is NP-complete.

Solution:
Let A be in P and let A not be the "empty" nor the "full" language. To show that A is NP-complete, we need to show that A is in NP (which it is, since $\mathsf{P} \subseteq \mathsf{NP}$) and that for every $L \in \mathsf{NP}$: $L \leq_p A$.
Since A is neither the empty language nor the full language, there exist two words x and x' such that $x \in A$ and $x' \notin A$. Let L be in NP. Since $\mathsf{P} = \mathsf{NP}$, there exists a polynomial-time TM M which decides L. The following polynomial-time computable function f reduces L to A: On input w, f simulates M on w. If M accepts then f outputs x, whereas if M rejects, f outputs x'.
We thus have $w \in L$ if and only if M accepts w if and only if $f(w) \in A$, which shows that f reduces L to A. Moreover, f is computable in polynomial time in the length of w, since M runs in polynomial time in the length of w. The function f thus shows that $L \leq_p A$.

For any language $\mathcal{L} \subseteq \Sigma^*$ we can define the complement of $\mathcal{L}$, $\overline{\mathcal{L}} = \{x \in \Sigma^*, x \notin \mathcal{L}\}$. In the case of a decision problem, we define the notion of complementation as follows: given a decision problem D, the complement of D is the problem whose answer is "yes" whenever D's answer is "no", and vice versa. For example, the complement of the 3-SAT problem is: given a 3-CNF formula ϕ, is it *not* satisfiable – i.e. is it the case that there is no x such that $\phi(x) = 1$? Note that the language corresponding to the complement of 3-SAT is not quite the same as the complement of the language corresponding to 3-SAT. The former is the set

of unsatisfiable 3-CNF formulae, while the latter is the set of bit-strings which do not encode satisfiable 3-CNF formulae. Usually this distinction will not be important.

For any complexity class C, we can thus define the complexity class $\mathsf{co\text{-}C}$ as the set $\{\overline{\mathcal{L}} : \mathcal{L} \in \mathsf{C}\}$. It is usually not the case that $\mathsf{co\text{-}C}$ is the complement of C. Indeed, it is immediate that, for example, $\mathsf{co\text{-}P} = \mathsf{P}$: a polynomial-time algorithm $\mathcal{A}$ for deciding $\mathcal{L}$ can be modified to give a polynomial-time algorithm $\overline{\mathcal{A}}$ for deciding $\overline{\mathcal{L}}$ by inverting the output of $\mathcal{A}$. However, in the case of nondeterministic classes the question is not so easy: in particular, it is not known whether $\mathsf{co\text{-}NP} = \mathsf{NP}$. Intuition behind this can be gained from the 3-SAT problem discussed above. Given a satisfiable formula ϕ, a satisfying assignment x gives a certificate that it is indeed satisfiable, which can be checked efficiently. On the other hand, if ϕ is not satisfiable by any assignment x, it is not clear how to exhibit a certificate of this which can be checked efficiently.

There is an alternative "operational" definition of $\mathsf{co\text{-}NP}$ which corresponds more closely (and perhaps intuitively) to our original definition of NP.

Definition 7. *For every $\mathcal{L} \subseteq \{0,1\}^*$, we say that $\mathcal{L} \in \mathsf{co\text{-}NP}$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that, for all $x \in \{0,1\}^*$, $x \in \mathcal{L}$ if and only if for all $u \in \{0,1\}^{p(|x|)}$, $M(x,u) = 1$.*

Note the change from "there exists" to "for all". We now check that these two definitions are indeed equivalent. If $\mathcal{L}$ satisfies Definition 7, then $x \in \overline{\mathcal{L}}$ if and only if there exists a u such that $M(x,u) = 0$; hence $\overline{\mathcal{L}} \in \mathsf{NP}$ (the NP verifier simply runs M and inverts the result). But if $\overline{\mathcal{L}} \in \mathsf{NP}$, there exists M' such that $x \in \mathcal{L} \Leftrightarrow M'(x,u) = 0$ for all u, hence $\mathcal{L} \in \mathsf{co\text{-}NP}$ by running M' and inverting the result.

Just as with NP, $\mathsf{co\text{-}NP}$ has complete problems, i.e. languages $\mathcal{L} \in \mathsf{co\text{-}NP}$ such that for all $\mathcal{L}' \in \mathsf{co\text{-}NP}$, $\mathcal{L}' \leq_{\mathsf{P}} \mathcal{L}$. A natural example is the problem Tautology: given a boolean formula ϕ, expressed in **DNF**, is ϕ satisfied by *every* assignment to its variables? Clearly, ϕ is a tautology if and only if $\neg\phi$ is not satisfiable.

Lemma 26. Tautology *is* $\mathsf{co\text{-}NP}$*-complete.*

Proof. We need to show that, for $\mathcal{L} \in \mathsf{co\text{-}NP}$, $\mathcal{L} \leq_{\mathsf{P}}$ Tautology. If $\mathcal{L} \in \mathsf{co\text{-}NP}$, there is a polynomial-time reduction R from $\overline{\mathcal{L}}$ to SAT, so for all $x \in \{0,1\}^*$, $x \in \overline{\mathcal{L}}$ if and only if $R(x)$ is satisfiable. Hence $x \in \mathcal{L}$ if and only if $\neg R(x)$ is satisfied by every assignment to its variables. Observe that, using De Morgan's law, $\neg R(x)$ can be written in DNF as required[1] by our definition of Tautology. □

We have the following characterisation of all $\mathsf{co\text{-}NP}$-complete problems.

Theorem 27. $\mathcal{L}$ *is* $\mathsf{co\text{-}NP}$*-complete if and only if* $\overline{\mathcal{L}}$ *is* NP*-complete.*

[1]The alert reader may have spotted that, just as SAT remains NP-complete if we relax the restriction that the input formula is in CNF, Tautology remains $\mathsf{co\text{-}NP}$-complete if we allow input formulae which are not in DNF.

Proof. We will show that, if $\overline{\mathcal{L}}$ is NP-complete, $\mathcal{L}$ is co-NP-complete (the other direction is similar). If $\overline{\mathcal{L}} \in \mathsf{NP}$, $\mathcal{L} \in \mathsf{co\text{-}NP}$ by definition. We also need to show that if $\overline{\mathcal{L}}$ is NP-hard, then $\mathcal{L}$ is co-NP-hard. Consider an arbitrary language $\mathcal{A} \in \mathsf{co\text{-}NP}$. As $\overline{\mathcal{L}}$ is NP-hard, and $\overline{\mathcal{A}} \in \mathsf{NP}$, there exists a polynomial-time computable function $R : \{0,1\}^* \to \{0,1\}^*$ such that $x \in \overline{\mathcal{A}} \Leftrightarrow R(x) \in \overline{\mathcal{L}}$. Thus $x \in \mathcal{A} \Leftrightarrow R(x) \in \mathcal{L}$, so $\mathcal{L}$ is co-NP-hard. □

Could it be possible that $\mathsf{NP} = \mathsf{co\text{-}NP}$? At the moment, we cannot rule this out, though such a result would seem almost as remarkable as $\mathsf{P} = \mathsf{NP}$: it would (informally) imply that any problem which has a short certificate of having a positive solution must also have a short certificate of having a negative solution. It is also unknown whether or not $\mathsf{P} = \mathsf{NP} \cap \mathsf{co\text{-}NP}$; while this would also be surprising, there are very few examples of problems known to be in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ but not known to be in P.

Consider the problem

$$FACTORING := \{(N, M) \mid N \text{ has a prime factor } k \text{ with } 1 < k < M\} \ .$$

This is clearly in NP as a witness can be an integer k) with k prime, $k < M$ and k dividing N. We can check primality, inequality and division in polytime, so we can clearly check the witness in polytime.

However, the problem also lies in co-NP as we now show. We have to show there is a witness w such that N can be verfied to have no prime factors of size less than M. For w we take the list of prime factors of N, there are at most $\log N$ of these, and they each have size bounded by $\log N$. So the witness is of polynomial size and we only need to check (again) primality, inequality and division (but now for $\log N$ numbers).

Self Test Question:
Prove that $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{co\text{-}NP}$, and that if $\mathsf{P} = \mathsf{NP}$, $\mathsf{NP} = \mathsf{co\text{-}NP}$.

Solution:
For the first part, it suffices to show that $\mathsf{P} \subseteq \mathsf{NP}$, and $\mathsf{P} \subseteq \mathsf{co\text{-}NP}$. The first of these inclusions was shown in Section 5.2. The proof of the second is essentially the same: for any language $\mathcal{L} \in \mathsf{P}$, given an empty certificate u, membership in $\mathcal{L}$ can be decided by ignoring u and running the polynomial-time algorithm for deciding membership in $\mathcal{L}$. For the second part, observe that $\mathsf{co\text{-}P} = \mathsf{P}$, so if $\mathsf{P} = \mathsf{NP}$, $\mathsf{NP} = \mathsf{co\text{-}NP}$.

3.2 The polynomial hierarchy

We have seen that the classes NP and co-NP have natural definitions in terms of "there exists" and "for all" quantifiers, respectively:

- $\mathcal{L} \in \mathsf{NP}$: $x \in \mathcal{L}$ if and only if $\exists\, w \in \{0,1\}^{\mathrm{poly}(|x|)}, M(x,w) = 1$.
- $\mathcal{L} \in \mathsf{co\text{-}NP}$: $x \in \mathcal{L}$ if and only if $\forall\, w \in \{0,1\}^{\mathrm{poly}(|x|)}, M(x,w) = 1$.

If we combine these two types of quantifier, we get a potentially more general set of complexity classes known as the polynomial hierarchy[2].

Definition 8. *For every $\mathcal{L} \subseteq \{0,1\}^*$, and for integer $i \geq 1$, we say that $\mathcal{L} \in \Sigma_i$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that, for all $x \in \{0,1\}^*$,*

$$x \in \mathcal{L} \Leftrightarrow \exists u_1 \in \{0,1\}^{p(|x|)} \, \forall u_2 \in \{0,1\}^{p(|x|)} \ldots \, Q_i u_i \in \{0,1\}^{p(|x|)}, M(x, u_1, \ldots, u_i) = 1,$$

where the quantifier Q_i is $\exists$ (if i is odd) or $\forall$ (if i is even). The polynomial hierarchy is the set $\mathsf{PH} = \bigcup_{i \geq 1} \Sigma_i$.

To help digest this definition, consider the following problem, called Largest Clique. Given a graph G and an integer k, is the largest clique within G of size exactly k? This problem is in the class Σ_2. To see this, observe that a graph has a largest clique of size k if and only if: there exists a set S of vertices such that S is of size k, and S is a clique; and for all sets S' of size $k+1$, S' is not a clique.

Recall that we showed that the problem Clique (given a graph G and an integer k, does G contain a clique on k vertices?) was NP-complete. However, if Largest Clique were contained within NP, there would exist an efficiently checkable certificate that there are no cliques in G of size at least $k+1$, and it is not clear what such a certificate would look like. Thus, similarly to the intuition that $\mathsf{NP} \neq \mathsf{co\text{-}NP}$, it is believed that $\Sigma_i \neq \Sigma_{i+1}$ for all i.

Self Test Question:
Let Smallest Formula be the following problem. Given a boolean formula ϕ in CNF, and an integer k, does there exist a CNF formula ϕ' such that ϕ' contains at most k symbols, and ϕ' computes the same function as ϕ? Prove that Smallest Formula $\in \Sigma_2$.

Solution:
The constraint is the same as "does there exist a formula ϕ' with at most k symbols such that $\phi'(x) = \phi(x)$ for all x?" This is clearly in Σ_2 as the verifier simply checks that ϕ' is a valid formula with at most k symbols and $\phi'(x) = \phi(x)$.

3.3 Decision vs Search

In practice we are often more interested in search problems than decision problems. However, Complexity Theory was originally built up using decision problems, as it came from the tradition of Turing Machines which focused on the decision problem of Halting. However, one can build up an entire theory of search problems analogous to our theory of decision problems. See the book by Goldreich for examples on how this can be done.

We can define analogues of the classes P and NP, which for this course we call $\mathsf{P}_{\mathsf{search}}$ and $\mathsf{NP}_{\mathsf{search}}$. To do this we look at relations.

[2]Short for "polynomial-time hierarchy".

Definition 9 (Relations)**.** *A relation R is a subset of the direct product of two sets $X \times Y$. If $(x, y) \in R$ we say x is related to y under R.*

For example if we take the standard relation $\leq$ then this is a subset of $\mathbb{R} \times \mathbb{R}$ and we have $(1, 2) \in \leq$ but $(2, 1) \notin \leq$.

Definition 10 (Poly-bounded Relations)**.** *A poly-bounded relation R is a subset of the direct product of $\{0,1\}^* \times \{0,1\}^*$, such that $|y| < p(|x|)$, for some polynomial p.*

Our search problems become elements of a poly-bounded relation. We think of x as the problem and y as the solution.

We can then define the class $\mathsf{P}_{\mathsf{search}}$ as follows:

Definition 11 (The class $\mathsf{P}_{\mathsf{search}}$)**.** *A poly bounded relation $R \in \mathsf{P}_{\mathsf{search}}$ if there is an algorithm A such that*

1. *For all x we have $A(x)$ outputs y.*
2. *We have $(x, y) \in R$, or $y = \perp$ and there is no y' such that $(x, y') \in R$.*
3. *A runs in poly time.*

Note, we "could" define P as the relations for which we can decide if there is such a y (a witness) in polytime. In which case we could have $\mathsf{P}_{\mathsf{search}} \subset \mathsf{P}$, since if we can find a witness in poly time we know it exists.

We define $\mathsf{NP}_{\mathsf{search}}$ as the set of search problems which have efficiently checkable solutions.

Definition 12 (The class $\mathsf{NP}_{\mathsf{search}}$)**.** *A poly bounded relation $R \in \mathsf{NP}_{\mathsf{search}}$ if there is an algorithm A such that*

1. *We have $(x, y) \in R$ if and only if $A(x, y) = 1$.*
2. *A runs in poly time.*

We clearly have $\mathsf{P}_{\mathsf{search}} \subset \mathsf{NP}_{\mathsf{search}}$.

An interesting class of problems are those for which we have a reduction from search to decision. Namely the search problem is no harder than a related decision problem. We present some examples in the lectures related to discrete logarithms and the knap-sack problem, below in the exercises we present one related to factoring.

Self Test Question:
Show that if $\mathsf{P} = \mathsf{NP}$ then you can factor integers in polynomial time Note that here you are asked to give an algorithm which *computes a function*, rather than *decides* a language; namely for the following problem:

FIND-FACTOR: Given an integer N, find $1 < k < N$ such that k divides N. If N is prime, return 1.

Solution:
In the notes we defined the following language:

$$FACTORING := \{(N, M) \mid N \text{ has an integer factor } k \text{ with } 1 < k < M\} \ .$$

Since we can divide integers in polynomial time, given (N, M) and k, we can easily check whether k divides N, and whether $1 < k < M$. Thus such a k is a certificate for $(N, M) \in$ *FACTORING*, and *FACTORING* is thus in NP. (We have seen this in the lecture.)
Now, if $\mathsf{P} = \mathsf{NP}$ then, since $FACTORING \in \mathsf{NP} = \mathsf{P}$, there exists a polynomial-time TM D_F which decides *FACTORING*. We now use D_F to construct a machine C_F which solves *FIND-FACTOR*, that is, C_F is given an integer N and if N is composite, it computes k with $1 < k < N$, such that k divides N; whereas if N is prime it outputs 1.
First note that if N has a factor then it has a factor less than $N/2$ (in fact, it even must have a factor less than $\sqrt{N}$, since if $N = ab$, we cannot have both $a < \sqrt{N}$ and $b < \sqrt{N}$). Thus, given N, C_F first runs D_F on input $(N, N/2)$. If D_F rejects, this means N has no factors (other than 1 and N), and is thus prime. C_F returns 1.
If D_F accepts then we know there is a factor $1 < k < N/2$. To find this factor, C_F splits the interval in two, that is, it runs D_F on $(N, N/4)$ to find out whether the factor lies between 1 and $N/4$ or between $N/4$ and $N/2$. If D_F accepts, C_F runs D_F on $(N, N/8)$, whereas if it rejects, C_F runs D_F on $(N, (3N)/8)$. Continuing in this way, in every step C_F halves the interval in which it knows a factor lies. Thus after $\log N$ rounds, it will determine a factor.
If D_F runs in polynomial time in the length of its input (N, M) then it runs in polynomial time in $n := \log N$, say $p(n)$. The running time of C_F is then $(\log N) \cdot p(n)$, which is polynomial in its input length n

Self Test Question:
Show that a polynomial-time algorithm for *FIND-FACTOR* implies a polynomial-time algorithm for *FACTORING*. (Hint: note that the number of prime factors of N is always less than $\log N$; and N has a factor smaller than M if and only if the smallest prime factor of N is smaller than M.)

Solution:
Now suppose there exists a polynomial-time TM C_F which solves *FIND-FACTOR*. We show that we can decide *FACTORING* in polynomial time by constructing a TM D_F.
First note that D_F should accept iff the *smallest prime factor* of N is less than M (since then and only then N has an integer factor less than M). However C_F is only guaranteed to return *any* (not necessarily prime) factor.
Moreover, note that the number d of prime factors of N is less than $\log N$. (Since if $N = p_1^{a_1} \cdots p_d^{a_d}$ is the prime factorisation of N then each $p_i \geq 2$ and $a_i \geq 1$, so $N \geq 2 \cdot 2 \cdots 2 = 2^d$, and thus $d \leq \log N$.)
D_F now uses C_F to find *all* prime factors of N and then checks whether there is one smaller than M: Run C_F on N. If it returns 1 (i.e. N is prime) then reject. If it returns k then run C_F on k. If this returns $k' \neq 1$ then run C_F on k', and so on, until we reach a prime p (that is, the first number on which C_F returns 1). We will use at most $\log N$ steps for this, since in every step the output number decreases by at least a factor of 2 (since $k < N/2$, $k' < k/2$, etc.). Thus finding a prime factor of N can be done in polynomial time.
Now D_F computes N/p, N/p^2, N/p^3, ..., as long as the powers of p divide N exactly. Let the last integer be $N' := N/p^\ell$. N' is thus N with the prime factor p removed. Again this can be done in polynomial time.
Next D_F repeats the above for N' to find the next prime factor p' of N, divides N' by the highest power of p' to get N'', which is N with two of its prime factors removed. Continuing this way until $N''^{\cdots'}$ is reduced to 1, D_F will have found *all* prime factors $p, p', p'', \ldots$ of N. Since there are at most $\log N$ such prime factors, finding all of them is performed in polynomial time. Now it only remains to check whether there is a prime factor less than M to decide *FACTORING*.

3.4 Worst case vs Average Case

Just like the case of search vs decision, traditional complexity theory also only deals with worst case analysis. But many problems, in fact it would appear from experiments most problems, are actually easy on average. Coming up with hard instances, is in in most cases quite hard (or you have to go out of your way to come up with a hard instance). For example most integers can be easily factored, since half are even! In industry SAT solvers are used routinely to solve SAT instances with thousands of variables.

When we looked at reductions we looked at relations such as $A \leq_P B$. This said that problem A was no harder than problem B. To show this we provided an algorithm (a poly-time reduction) which allowed us to solve problem A using an algorithm to solve B. We could then use any algorithm to solve B to also solve A.

It is clear that if we concentrate on the same problem then we have

$$\text{Adverage Case Problem } A \leq_P \text{ Worst Case Problem } A,$$

since if we have an efficient algorithm to solve the worst case we can always use it directly on an instance of an average case. What we *would like* is for problems to have the opposite

relation, i.e.

$$\text{Worst Case Problem } A \leq_P \text{ Average Case Problem } A.$$

In other words an algorithm which takes a worst case instance and turns it into an average instance. Such a reduction is called a *randomized self reduction*.

There are very few problems for which we know a randomized self reduction. Depending on the problem the existance of such a reduction implies either all instances of the problem are easy *or* all instances are hard. As an example let us take the *Decision Diffie–Hellman problem*.

Definition 13 (Decision Diffie–Hellman)**.** *Let G be a finite abelian group of prime order q. Consider the following problem instance generator:*

- *Pick $A \leftarrow G$, note A is a random generator.*
- *Pick $x, y, z \leftarrow [0, q-1]$.*
- *Pick $b \leftarrow \{0, 1\}$.*
- *Set $B = A^x, C = A^y$.*
- *If $b = 0$ set $D = A^{xy}$ else set $D = A^z$.*

Output (A, B, C, D). The problem is to decide whether the instance generator picked $b = 0$ or $b = 1$.

This problem is clearly in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$. We now show how to construct an algorithm which takes a worst-case instance and produces an average case instance, whose solution is the same as the worst-case instance we started with.

Let (A, B, C, D) be the input (supposedly worst case) instance. To get an average case instance we need to respect the distribution produced by the instance generator above, i.e. we need completely random and independent A', B', C' such that the resulting D' has the same properties as the D from the input instance. Here is how the randomization works

- $r_1, r_2, r_3 \leftarrow [0, q-1]$.
- $A' = A^{r_1}, B' = B^{r_1 \cdot r_2}, C' = C^{r_1 \cdot r_3}$.
- $D' = D^{r_1 \cdot r_2 \cdot r_3}$.

Note, that due to the use of r_1, r_2 and r_3 the values of A', B', C' are uniformly generated over the group G, just like the instance generator produces. We have

$$A' = A^{r_1}, \quad B' = A^{x \cdot r_1 \cdot r_2} = A'^{x \cdot r_2}, \quad C' = A^{y \cdot r_1 \cdot r_3} = A'^{y \cdot r_3}.$$

Thus we have $x' = x \cdot r_2 \pmod q$ and $y' = y \cdot r_3 \pmod q$. Now look what happens when the original instance was such that $b = 0$ and $D = A^{xy}$

$$D' = A^{x \cdot y \cdot r_1 \cdot r_2 \cdot r_3} = A'^{x \cdot y \cdot r_2 \cdot r_3} = A'^{x' \cdot y'}$$

i.e. the solution is still $b = 0$. However, when $b = 1$ and $D = A^z$ we have

$$D' = A^{z \cdot r_1 \cdot r_2 \cdot r_3} = A'^{z \cdot r_2 \cdot r_3} \neq A'^{x' \cdot y'}$$

i.e. the solution is still $b = 1$.

3.5 Supplementary Notes

Self Test Question:
Prove that, if $\mathsf{P} = \mathsf{NP}$, $\mathsf{EXP} = \mathsf{NEXP}$.

Solution:
The basic idea is to pad the input. For any $\mathcal{L} \in \mathsf{NTIME}(2^{n^c})$, the language $\mathcal{L}' = \{(x, 0^{2^{|x|^c}}) : x \in \mathcal{L}\}$ is in $\mathsf{NTIME}(n)$. Assuming that $\mathsf{P} = \mathsf{NP}$, this implies that $\mathcal{L}' \in \mathsf{DTIME}(n^d)$ for some d. For any x, to determine whether $x \in \mathcal{L}$ it suffices to determine whether $(x, 0^{2^{|x|^c}}) \in \mathcal{L}'$, which can be done in time $O(2^{d|x|^c})$, so $\mathcal{L} \in \mathsf{EXP}$.

3.6 Advanced Technical Notes

Just as with NP and $\mathsf{co\text{-}NP}$, each level of the polynomial hierarchy has a natural complete problem, a "quantified" variant of SAT. Let ϕ be a boolean formula on n variables $x_1, \ldots, x_n$ partitioned into sets $X_1, \ldots, X_i$. Then a QSAT_i instance is an expression

$$\exists X_1 \, \forall X_2 \, \exists X_3 \, \ldots Q_i X_i \, \phi,$$

where Q_i is $\exists$ (if i is odd) or $\forall$ (if i is even). For example, the following expression is a QSAT_3 instance:

$$\exists x_1, x_2 \, \forall x_3 \, \exists x_4, x_5 \, (x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_4).$$

The QSAT_i problem is then simply to determine whether such an expression evaluates to true. Observe that QSAT_1 is just the SAT problem. More generally, we have the following theorem, which we will not prove but should not be too surprising given the definitions of QSAT_i and Σ_i.

Theorem 28. *$QSAT_i$ is Σ_i-complete.*

This allows us to give the following alternative and concise definition of the polynomial hierarchy in terms of oracle machines.

Theorem 29. $\Sigma_1 = \mathsf{NP}$, *and for* $i \geq 2$, $\Sigma_\mathsf{i} = \mathsf{NP}^{QSAT_{i-1}}$.

Proof idea. To see the essential ideas, we will prove the special case that $\Sigma_2 = \mathsf{NP}^{\mathrm{SAT}}$.

- ($\Sigma_2 \subseteq \mathsf{NP}^{\mathrm{SAT}}$.) Let $\mathcal{L} \in \Sigma_2$. Then there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that, for all $x \in \{0,1\}^*$,

$$x \in \mathcal{L} \Leftrightarrow \exists u_1 \in \{0,1\}^{p(|x|)} \, \forall u_2 \in \{0,1\}^{p(|x|)}, M(x, u_1, u_2) = 1.$$

Our nondeterministic algorithm with access to an oracle for SAT will proceed as follows. On input x, the algorithm guesses a string u_1, and would like to determine whether there exists $u_2 \in \{0,1\}^{p(|x|)}$ such that $M(x, u_1, u_2) = 0$. By the Cook-Levin theorem, this can be written as a SAT instance. The algorithm therefore uses its oracle for SAT, which outputs YES if there exists $u_2 \in \{0,1\}^{p(|x|)}$ such that $M(x, u_1, u_2) = 0$, and accepts if and only if the answer is NO. Thus it accepts if and only if $\exists u_1 \in \{0,1\}^{p(|x|)} \forall u_2 \in \{0,1\}^{p(|x|)}, M(x, u_1, u_2) = 1$.

- ($\mathsf{NP}^{\mathrm{SAT}} \subseteq \Sigma_2$.) Let $\mathcal{L} \in \mathsf{NP}^{\mathrm{SAT}}$, and let M be an NDTM which decides $\mathcal{L}$ with access to an oracle for SAT. We first show that there is a machine N which also decides $\mathcal{L}$, makes one query to the oracle, and accepts if and only if the answer to the oracle query is NO. On input x, N guesses all of the nondeterministic choices of M, all of the oracle questions, and all of the answers. For each question that was answered with YES, N guesses a satisfying assignment to check that the guess was correct. At the end, N still has a set of oracle questions (i.e. formulae) $\phi_1, \ldots, \phi_k$ to which the oracle would have replied with NO. N then asks the oracle whether the formula $\phi_1 \vee \cdots \vee \phi_k$ is satisfiable, and accepts if and only if the answer is NO.

  We now observe that N accepts (i.e. $x \in \mathcal{L}$) if and only if there exist a set of nondeterministic choices, oracle questions and answers, such that there is no satisfying assignment to $\phi_1 \vee \cdots \vee \phi_k$. Thus $\mathcal{L} \in \Sigma_2$ as required.

□

Papadimitriou chapter 17 has an extensive discussion of the polynomial hierarchy, which was defined by Meyer and Stockmeyer in 1972; there are also some good lecture notes by Luca Trevisan[3].

Self Test Question:
Prove that $\mathsf{co\text{-}NP} \subseteq \mathsf{P}^{\mathrm{SAT}}$.

Solution:
It suffices to show that a Turing machine running in polynomial time equipped with an oracle for SAT can solve TAUTOLOGY. Given a DNF formula ϕ, our algorithm proceeds as follows. It takes $\neg\phi$, converts it into CNF using De Morgan's law, and uses the oracle for SAT to determine whether $\neg\phi$ is satisfiable. The algorithm then inverts the answer and outputs the result.

Self Test Question:
Prove that, if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{PH} = \mathsf{P}$ – "the polynomial hierarchy collapses".

Solution:
Define the class Π_i in terms of i levels of alternating quantifiers in the same way as Σ_i, but starting with a $\forall$ quantifier, rather than $\exists$. To prove the claim, it suffices to show by induction that $\Pi_i = \mathsf{P}$ implies that $\Sigma_{i+1} = \mathsf{P}$, and similarly that $\Sigma_i = \mathsf{P}$ implies that $\Pi_{i+1} = \mathsf{P}$. Assuming that $\Pi_i = \mathsf{P}$, we prove that $\Sigma_{i+1} = \mathsf{P}$ (the second part is similar). $\mathcal{L} \in \Sigma_{i+1}$ implies that $x \in \mathcal{L}$ if and only if $\exists u_1 \forall u_2 \ldots Q_{i+1} u_{i+1} M(x, u_1, \ldots, u_{i+1}) = 1$. By definition, the language $\mathcal{L}'$ defined by $(x, u_1) \in \mathcal{L}'$ if and only if $\forall u_2 \ldots Q_{i+1} u_{i+1} M(x, u_1, \ldots, u_{i+1}) = 1$ is in Π_i. By the inductive hypothesis, $\mathcal{L}' \in \mathsf{P}$, so $\mathcal{L} \in \mathsf{NP}$. As we have assumed that $\mathsf{P} = \mathsf{NP}$, $\mathcal{L} \in \mathsf{P}$, implying $\Sigma_{i+1} = \mathsf{P}$

[3] `http://www.cs.berkeley.edu/~luca/cs278-08/lecture04.pdf`

Self Test Question:
Give an *explicit* algorithm for SAT which, given a boolean formula ϕ, outputs a satisfying assignment for ϕ in polynomial time, assuming that $\mathsf{P} = \mathsf{NP}$. If there is no satisfying assignment, the algorithm can behave arbitrarily. [HARD]

Solution:
We loop over all Turing machines. At stage s, we simulate each Turing machine $M_1, \ldots, M_s$ indexed by $1, \ldots, s$ for s steps. If one of these M_i outputs a satisfying assignment, terminate. If $\mathsf{P} = \mathsf{NP}$, there exists a Turing machine M_m that runs in time at most $|\phi|^k$, for some k, and outputs a satisfying assignment for ϕ (see Theorem 7.2). This machine will be run for time $|\phi|^k$ at stage $\max\{m, |\phi|^k\}$. Stage s uses time $\text{poly}(|\phi|)\,\text{poly}(s)$. As m is constant (!), the whole algorithm runs in time polynomial in $|\phi|$.

Lecture 4

Space Complexity

So far we have studied computations which are restricted with respect to the time that they use. Another important resource to consider is space. In the Turing machine model, this corresponds to the number of cells used on the tape. Arora-Barak has a general discussion of space complexity in chapter 4. The Immerman-Szelepcsényi Theorem was proven independently by Immerman and Szelepcsényi in the late 80's. The fact that the PATH problem restricted to undirected graphs is in L was shown by Reingold in 2005; Arora-Barak chapter 21 includes a proof.

4.1 Space complexity

In particular, we would like to study computations which use less space than the size of the input. In order for this to make sense, we modify the Turing machine model as follows.

Definition 14. *A space-bounded (deterministic/nondeterministic) Turing machine M is a multi-tape Turing machine with a special read-only input tape, on which the input x is placed prior to the start of the computation. All other tapes are known as work tapes. The output $M(x)$ is placed on one of these work tapes, which is known as the output tape.*

Let $s : \mathbb{N} \to \mathbb{N}$ and $\mathcal{L} \subseteq \{0,1\}^*$. We say that $\mathcal{L} \in \mathsf{SPACE}(s(n))$ if there is a constant $c > 0$ and a deterministic Turing machine M which decides $\mathcal{L}$, such that ;at most $c\,s(n)$ locations on M's work tapes are ever scanned by M's head during the computation, on any input of size n. Similarly, $\mathcal{L} \in \mathsf{NSPACE}(s(n))$ if there is an NDTM M which decides $\mathcal{L}$, under the same constraints for any of M's nondeterministic choices. We usually restrict to *space-constructible* functions $s(n)$, i.e. functions $s(n)$ for which there exists a Turing machine that computes $s(|x|)$ in space $O(s(|x|))$ on input x. As with time-constructible functions, this is a very minor restriction as all "sensible" functions (e.g. $\lceil \log n \rceil$, n, 2^n) are space-constructible.

By analogy with their time-complexity equivalents, we define $\mathsf{PSPACE} = \bigcup_{c>0} \mathsf{SPACE}(n^c)$, $\mathsf{NPSPACE} = \bigcup_{c>0} \mathsf{NSPACE}(n^c)$. Two classes which do not have a sensible time-complexity analogue are

$$\mathsf{L} = \mathsf{SPACE}(\log n),\ \mathsf{NL} = \mathsf{NSPACE}(\log n).$$

Observe that in these cases the space used is significantly less than the size of the input; the corresponding time complexity classes would correspond to algorithms which do not have time to read the whole input, which are usually uninteresting[1]. We can thus imagine an L computation as one where the size of the data on which the computer is operating is much larger than the size of the computer itself. An intuitive example of this phenomenon from daily experience is the use of the Internet: the Internet is too large to be stored within our computers, but we can access it and retrieve data, on which we can perform computations.

As with the case of NP, the class NL has an alternative definition in terms of certificates, which we formalise as follows.

Definition 15. NL *is the class of languages* $\mathcal{L} \subseteq \{0,1\}^*$ *such that there exists a polynomial* $p : \mathbb{N} \to \mathbb{N}$ *and a space-bounded Turing machine* M *such that:*

- M *has an additional certificate tape which contains a witness* w*. The certificate tape is read-only and scanned once, left to right;*
- *for all* $x \in \{0,1\}^*$*,* $x \in \mathcal{L}$ *if and only if there exists* $w \in \{0,1\}^{p(|x|)}$ *such that* $M(x,w) = 1$*;*
- M *uses at most* $O(\log |x|)$ *space on its work tapes for all* x*.*

It should be clear that this alternative definition is equivalent, as making a sequence of nondeterministic choices can be viewed as reading the bits of a certificate one by one, in order. The "read-once" restriction is necessary: if it is removed, the class of languages decided by such machines becomes equal to NP.

We observe that for any $f : \mathbb{N} \to \mathbb{N}$, $\mathsf{DTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$, as no machine using T computational steps can visit more than T tape cells. However, we can say somewhat more.

Theorem 30. *For any function* $f : \mathbb{N} \to \mathbb{N}$ *such that* $f(n) \geq \log_2 n$*,*

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))}).$$

Proof. The first and third inclusions are obvious. The second is based on the exponential-time simulation of NDTMs by DTMs given in Theorem 12. Given an NDTM with two transition functions that runs in time T, we simulate it by a DTM which tries each of the possible 2^T sequences of choices of transition functions in turn. To iterate through each of these transition functions requires only $O(T)$ space (to store a counter).

To prove the final inclusion, we use the concept of a *configuration graph*. For every space $s(n)$ deterministic or nondeterministic Turing machine M and input $x \in \{0,1\}^*$ such that $|x| = n$, the configuration graph $G_{M,x}$ of M on input x is the directed graph whose nodes are all possible configurations c of M such that the input is x and the work tapes have at most $s(n)$ non-blank cells in total. $G_{M,x}$ has an arc $c \to c'$ if M's transition function can go from c to c'. If M is deterministic, $G_{M,x}$ has out-degree 1 (i.e. is a path), and if M is

[1]However, at the end of these notes we will see a model in which this idea does make sense.

nondeterministic, $G_{M,x}$ has out-degree at most 2. Let the starting configuration be c_S, and assume that there is a unique configuration c_A on which M accepts (this is without loss of generality as we can modify M to erase the contents of all its work tapes before accepting). Then M accepts x if and only if there is a path from c_S to c_A.

Observe that, for some constant c (depending on M) $G_{M,x}$ has at most $n2^{c\,s(n)}$ nodes, as a configuration is completely defined by M's state, the head positions[2] and the contents of the work tapes. Therefore, by enumerating all possible configurations, the graph $G_{M,x}$ can be constructed in time $2^{O(s(n))}$. We can then test whether there is a path from c_S to c_A using the algorithm for PATH, which runs in time $\text{poly}(2^{O(s(n))}) = 2^{O(s(n))}$. □

We also have the following analogue of the Time Hierachy Theorem (Theorem 1) for space complexity.

Theorem 31 (Space Hierarchy Theorem)**.** *If f and g are space-constructible functions such that $f(n) = o(g(n))$, then* $\mathsf{SPACE}(f(n)) \subset \mathsf{SPACE}(g(n))$.

The proof is the same as for the Time Hierarchy Theorem, but note that the result is somewhat tighter. The reason is that it is possible to design a universal Turing machine with only a constant factor space overhead.

4.2 Polynomial space

Intuitively, space appears to be a more powerful resource than time, as space can be reused whereas time cannot. Remarkably, it is nevertheless an open question whether $\mathsf{P} = \mathsf{PSPACE}$. However, as (by Theorem 30), we know that $\mathsf{NP} \subseteq \mathsf{PSPACE}$, it seems unlikely that this equality holds. In fact, it turns out that we can solve problems even more general than SAT in PSPACE.

A quantified boolean formula is a formula of the form

$$Q_1 x_1\, Q_2 x_2\, \ldots Q_n x_n\, \phi(x_1, \ldots, x_n),$$

where each $Q_i \in \{\forall, \exists\}$, and ϕ is a standard boolean formula. The TQBF problem ("true/totally quantified boolean formula") is defined as follows: given a quantified boolean formula ψ, determine whether ψ is true. Observe that this is a generalisation of the QSAT_i problem which we previously encountered: rather than the number of quantifiers being at most i, for some fixed i, now there can be up to n quantifiers. The above formula has all its quantifiers at the start (so-called "prenex normal form"). One could conceive of more general quantified boolean formulae with quantifiers appearing throughout the formula, e.g.

$$\exists x_1\, (x_1 \vee \forall x_2, x_3(\neg x_2 \wedge x_3)).$$

[2]The "n" comes from the input tape head position. As $s(n) \geq \log_2 n$, this can be absorbed into $2^{O(s(n))}$.

In fact, this is not a generalisation as all such formulae can be converted into prenex normal form in polynomial time. The quantifiers $Q_1, \ldots, Q_n$ may be assumed to alternate between $\forall$ and $\exists$ by introducing dummy variables. That is,

$$\exists x_1 \, \exists x_2 \, \phi(x_1, x_2) = \exists x_1 \, \forall y_1 \, \exists x_2 \, \phi(x_1, x_2),$$

etc. Observe that SAT is the special case of TQBF where each quantifier $Q_i = \exists$. That is, a boolean formula ϕ is satisfiable if and only if

$$\exists x_1 \, \exists x_2 \, \ldots \exists x_n \, \phi(x_1, \ldots, x_n).$$

Theorem 32. *TQBF is* PSPACE*-complete.*

In the statement of this theorem, PSPACE-completeness is used exactly in the sense that NP-completeness was before. Explicitly, we say that a language $\mathcal{A}$ is PSPACE-complete if $\mathcal{A} \in \mathsf{PSPACE}$, and for all $\mathcal{B} \in \mathsf{PSPACE}$, $\mathcal{B} \leq_{\mathsf{P}} \mathcal{A}$.

Proof. We first show that TQBF is in PSPACE. Let $\psi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \, \phi(x_1, \ldots, x_n)$ be a quantified boolean formula, and let the size of ϕ be m. We give a recursive algorithm $\mathcal{A}$ which can decide whether ψ is true in space $\mathrm{poly}(n, m)$.

For $b \in \{0, 1\}$, let $\psi_{x_1=b}$ denote the quantified boolean formula obtained by removing the first quantifier Q_1 and replacing x_1 with b throughout the formula ϕ. $\mathcal{A}$ will do the following.

- If $Q_1 = \exists$, then output 1 if and only if at least one of $\mathcal{A}(\psi_{x_1=0})$ and $\mathcal{A}(\psi_{x_1=1})$ output 1.
- If $Q_1 = \forall$, then output 1 if and only if both of $\mathcal{A}(\psi_{x_1=0})$ and $\mathcal{A}(\psi_{x_1=1})$ output 1.

It is clear that $\mathcal{A}$ works correctly; we now show that $\mathcal{A}$ can be implemented to use only polynomial space. Let $s_{n,m}$ be the space used by $\mathcal{A}$ on formulae with n variables and total size m. After computing $\mathcal{A}(\psi_{x_1=0})$, the algorithm can reuse the same space to compute $\mathcal{A}(\psi_{x_1=1})$, needing only to retain one bit to store the value of $\mathcal{A}(\psi_{x_1=0})$. At each step the algorithm also needs to use only $O(m)$ space to store a description of $\psi_{x_1=0}$ and $\psi_{x_1=1}$, to be passed to the next recursive call of $\mathcal{A}$. Thus $s_{n,m} = s_{n-1,m} + O(m)$, so $s_{n,m} = O(nm)$.

We now show that, for any $\mathcal{L} \in \mathsf{PSPACE}$, $\mathcal{L} \leq_{\mathsf{P}}$ TQBF. The proof will be based on the idea of configuration graphs which we saw previously. We are given a Turing machine M that decides $\mathcal{L}$ in space $s(n)$. For each $x \in \{0, 1\}^n$, we aim to construct, in polynomial time, a quantified boolean formula ψ of size $O(s(n)^2)$ which is true if and only if M accepts x. We will achieve this by producing a sequence of quantified formulae ψ_i such that $\psi_i(a, b)$ is true if and only if there is a path of length at most 2^i in $G_{M,x}$ from a to b. We can then output $\psi_{O(s(n))}(c_S, c_A)$, where c_S, c_A are the start and accepting configurations of M. As the configuration graph of M has at most $2^{O(s(n))}$ nodes, this suffices to determine whether M accepts $\mathcal{L}$.

We first make the following claim: there is an $O(s(n))$-size CNF formula ϕ such that, for any two configurations c, c', $\phi(c, c') = 1$ if and only if there is an arc $c \to c'$ in $G_{M,x}$. To see

this, note that each configuration can be encoded as an $O(s(n))$-bit string and so, as in the proof of the Cook-Levin theorem, whether one configuration can follow from another can be encoded as the AND of at most $O(s(n))$ constant-sized checks. This implies that we can construct $\psi_0(a, b)$ by taking the OR of the formula ϕ and the formula which checks whether a and b are equal.

We now want to use this idea to construct ψ_{i+1}, given ψ_i. The intuitive way of doing this would be to write down the quantified formula

$$\exists c\, \psi_i(a, c) \wedge \psi_i(c, b),$$

which encodes the claim that (in words) "there is a path from a to b of length at most 2^{i+1} if and only if there exists some c such that there is a path from a to c of length at most 2^i, and a path from c to b of length at most 2^i". However, this approach would result in obtaining a formula of exponential size, as at each step ψ_i could double in size. We therefore instead define $\psi_{i+1}(a, b)$ as

$$\exists c \forall x, y : ((x = a \wedge y = c) \vee (x = c \wedge y = b)) \Rightarrow \psi_i(x, y),$$

which encodes the same claim (as one can verify). Then we have that the size of ψ_{i+1} is at most the size of ψ_i, plus $O(s(n))$. Therefore, $\psi_{s(n)}$ is of size $O(s(n)^2)$. It finally remains to convert the formula $\psi_{O(s(n))}$ to normal form, which can be done in polynomial time. □

TQBF remains PSPACE-complete even if we restrict the boolean formula ϕ to be in CNF. To see this, observe that for any x, y, "$x \Rightarrow y$" can be written as $\neg x \vee y$, implying that for each i, ψ_i can be written in DNF without significantly increasing in size. Taking the negation of the final formula $\psi_{s(n)}$ gives a CNF formula for the *complement* of the PSPACE language $\mathcal{L}$; but co-PSPACE = PSPACE, so the claim follows. The very observant reader may have noticed that the above proof that for any $\mathcal{L} \in$ PSPACE, $\mathcal{L} \leq_{\mathsf{P}}$ TQBF also works without change for nondeterministic Turing machines. This implies that (as TQBF $\in$ PSPACE) in fact PSPACE = NPSPACE! This is in stark contrast to our intuition that P $\neq$ NP.

4.3 Log space

The PSPACE vs. NPSPACE question has a log-space analogue: does L = NL? Unlike PSPACE vs. NPSPACE, this question is open, although some interesting partial results towards resolving it are known. We begin by giving a natural complete problem for NL.

Given that NL $\subseteq$ P by Theorem 30, we cannot use the same notion of polynomial-time reductions that we used before to sensibly define what it means for a problem to be NL-complete. The notion we will use instead is that of log-space reductions, the definition of which requires a little care.

We say that $f : \{0,1\}^* \to \{0,1\}^*$ is implicitly log-space computable if f is polynomially bounded (i.e. there exist b, c such that $|f(x)| \leq b|x|^c$ for all $x \in \{0,1\}^*$), and the languages $\mathcal{L}_f = \{(x, i) : f(x)_i = 1\}$ and $\mathcal{L}'_f = \{(x, i) : |f(x)| \geq i\}$ are in L. That is, given x, we

can determine any individual bit of $f(x)$, and the length of $f(x)$, using a log-space Turing machine.

We then say that $\mathcal{A}$ is log-space reducible to $\mathcal{B}$, and write $\mathcal{A} \leq_{\mathsf{L}} \mathcal{B}$, if there is an implicitly log-space computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in \mathcal{A}$ if and only if $f(x) \in \mathcal{B}$. We finally say that a language $\mathcal{A}$ is NL-complete if $\mathcal{A} \in \mathsf{NL}$, and for all $\mathcal{B} \in \mathsf{NL}$, $\mathcal{B} \leq_{\mathsf{L}} \mathcal{A}$.

Theorem 33. PATH *is* NL*-complete.*

Proof. We first show that PATH is in NL. If there is a path from node s to node t, a nondeterministic machine can find it by repeatedly guessing the neighbour of the current node which is on the path. This only requires the number of steps taken on the path to be known, and the identity of the current node. As the length of the path can be taken to be at most n, we only need $O(\log n)$ space to store this information.

Let $\mathcal{L} \in \mathsf{NL}$; we now show that $\mathcal{L} \leq_{\mathsf{L}}$ PATH. Let M be a log-space NDTM which decides $\mathcal{L}$, and for any input x of size n, consider the configuration graph $G_{M,x}$ (assuming as before that M has starting configuration c_S, and a single accepting configuration c_A). Our reduction will take the pair (M, x) as input and produce the adjacency matrix of $G_{M,x}$. As there is a path in this graph from c_S to c_A if and only if M accepts x, an algorithm for PATH can be used to determine from this adjacency matrix whether M accepts x. It remains to show that this reduction is implicitly log-space computable. This follows because, given a pair of configurations (c, c'), in space $O(|c| + |c'|) = O(\log n)$ a Turing machine can examine each of them and determine whether c' is one of the two configurations which can follow c, based on M's transition rules. □

By Theorem 33, resolving the $\mathsf{L} \stackrel{?}{=} \mathsf{NL}$ question is equivalent to determining whether there is a log-space algorithm for PATH. While it is not known whether such an algorithm exists, it has recently been shown that PATH restricted to *undirected* graphs is indeed in L. We also have the following result.

Theorem 34. PATH $\in \mathsf{SPACE}(\log^2 n)$.

Proof. The basic idea is very similar to the second part of the proof of Theorem 32. Let G be a directed graph with n nodes, and let $\text{PATH}_i(a, b)$ be the following problem: given G (as an adjacency matrix) and nodes a and b in G, is there a path from a to b of length at most 2^i? Observe that if we can solve $\text{PATH}_{\lceil \log_2 n \rceil}$, we can solve PATH, as no path need be of length longer than n. We solve PATH_i by a recursive technique. If $i = 0$, we can solve $\text{PATH}_0(a, b)$ simply by checking whether $a = b$, or there is an arc from a to b. If $i \geq 1$, we compute $\text{PATH}_i(a, b)$ as follows: output true if and only if there exists a node c such that $\text{PATH}_{i-1}(a, c)$ is true and $\text{PATH}_{i-1}(c, b)$ is true. To see that this is correct, observe that (as in the proof of Theorem 32) there is a path of length 2^i from a to b if and only if there is a path of length at most 2^{i-1} from a to some "midpoint" c, and a path of length at most 2^{i-1} from c to b.

It remains to implement this computation space-efficiently. But this can be done using exactly the same idea as in the first part of the proof of Theorem 32; we obtain that the space used is $O(\log^2 n)$. □

Theorem 34 has the following corollary, a generalisation of the result that $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Theorem 35 (Savitch's Theorem)**.** *For any function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq \log_2 n$, $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{SPACE}(s(n)^2)$.*

Proof. For any $\mathcal{L} \in \mathsf{NSPACE}(s(n))$, given an NDTM M which decides $\mathcal{L}$ in space $s(n)$, we simply run the algorithm of Theorem 34 on the configuration graph of M, which as discussed before has at most $2^{O(s(n))}$ nodes. □

Self Test Question:
Show that, if $\mathcal{A} \in \mathsf{NL}$ and $\mathcal{B}$ is any non-trivial language, $\mathcal{A} \leq_{\mathsf{P}} \mathcal{B}$. That is, almost every language $\mathcal{B}$ is NL-hard under polynomial-time reductions.

Solution:
As $\mathsf{NL} \subseteq \mathsf{P}$, our reduction from $\mathcal{A}$ to $\mathcal{B}$ can just decide membership in $\mathcal{A}$ in polynomial time, then output something which is in $\mathcal{B}$ (if the input was in $\mathcal{A}$) or not in $\mathcal{B}$ (if the input was not in $\mathcal{A}$).

Self Test Question:
Prove composability of log-space reductions: i.e. that, if $\mathcal{A} \leq_{\mathsf{L}} \mathcal{B}$ and $\mathcal{B} \leq_{\mathsf{L}} \mathcal{C}$, $\mathcal{A} \leq_{\mathsf{L}} \mathcal{C}$.

Solution:
We need to show that, if $f, g : \{0,1\}^* \to \{0,1\}^*$ are implicitly log-space computable, so is the composition $f \circ g$. If f and g are polynomially bounded, so is $f \circ g$. We also need to show that the languages $\mathcal{L}_{f \circ g}$ and $\mathcal{L}'_{f \circ g}$ (as defined in the notes) are in L. For the first of these (the second is similar) the idea is as follows. We want to compute the bit $(f \circ g)(x)_i$ given Turing machines M_f, M_g which compute $f(x)$, $g(x)$ for any x. M_f computes $f(g(x))$ by accessing the bits of $g(x)$. Each time M_f would like to access the bit $g(x)_j$, we simply compute this bit by simulating M_g. See Arora-Barak for more details (proof of Lemma 4.17).

Self Test Question:
Let $\mathcal{L}$ be the language of properly nested parentheses. For example, (()) and (()(()())) are in $\mathcal{L}$ but)(is not. Show that $\mathcal{L} \in \mathsf{L}$.

Solution:
We use a work tape to store a counter in binary. Our Turing machine M reads each input parenthesis symbol and increments the counter (if it is an open parenthesis symbol) or decrements it (if it is a close parenthesis symbol). If the counter is ever decremented below 0, we reject; if the input ends with the counter being nonzero, we reject. Otherwise, we accept. If the input is of size n, the counter will never use more than $O(\log n)$ bits of storage space.

Self Test Question:
Prove the claim in the lecture notes that, if the read-once restriction is removed from the definition of NL, the resulting class is equal to NP.

Solution:
One way to prove this is to give an algorithm which solves 3-SAT in logarithmic space, given (not just read-once) access to a certificate. Given a boolean formula ϕ, the certificate is a satisfying assignment x for ϕ. The algorithm checks whether each clause of ϕ in turn is satisfied by x. Checking each clause can be done using only constant workspace by checking each literal in turn and taking the OR of the three bits. Thus satisfiability of the whole formula can be checked using logarithmic workspace by checking each clause in turn, storing a counter on a work tape to keep track of which clause is currently being checked.
Also need to use log-space version of Cook-Levin!

4.4 Supplementary Notes

Another question about space complexity which has been resolved and stands in contrast to the corresponding time complexity conjecture is the NL vs. co-NL question.

Theorem 36 (Immerman-Szelepcsényi Theorem)**.** NL = co-NL.

The technique used to prove Theorem 36 is a direct attack: giving a nondeterministic log-space algorithm for the complement of the PATH problem. It is worth pausing to note that it is somewhat surprising that this works; while it is easy to certify that there is a path between two nodes s and t (just give the path), it is less clear how to certify that there is *no* path from s to t.

Proof of Theorem 36. The complement of the PATH problem, $\overline{\text{PATH}}$, may be defined as follows. Given a graph G with n nodes, and specified nodes s, t, output 1 if there is no path from s to t, and 0 otherwise. We prove that $\overline{\text{PATH}}$ is in NL by showing the existence of a certificate, which can be checked with read-once access in logarithmic space, that there is no path from s to t. The certificate will be built from a sequence of subsidiary certificates. Let C_i denote the set of nodes which can be reached from s in at most i steps, and write $c_i = |C_i|$; we want to certify that $t \notin C_n$. We know (see the proof of Theorem 33) that, for each i and any v, there exists a certificate $\text{Path}_i(s, v)$ which is read-once checkable in logarithmic space and certifies that there is a path of length at most i from s to v, i.e. proves that $v \in C_i$. We now use an inductive procedure to design two additional certificates.

1. A certificate that $v \notin C_i$, assuming that the verifier knows $|C_i|$.
2. A certificate that $|C_i| = c$ for some c, assuming that the verifier knows $|C_{i-1}|$.

As the verifier already knows that $C_0 = \{s\}$ at the beginning, using these two types of certificate for $i = 1, \ldots, n$ enables the verifier to determine that $t \notin C_n$. We now show how to construct them.

1. The certificate is simply the list of all nodes in C_i, together with their certificates that they are in C_i:

$$v_1, \text{Path}_i(s, v_1), \ldots, v_{c_i}, \text{Path}_i(s, v_{c_i})$$

for $v_1, \ldots, v_{c_i} \in C_i$ in ascending order. The verifier checks that (1) the number of nodes is equal to c_i; (2) the nodes are listed in ascending order; (3) none of the nodes is equal to v; (4) each certificate is valid. It is easy to see that all of these checks can be done with read-once access to the whole certificate. If $v \notin C_i$, such a certificate exists; however, if $v \in C_i$ no such list of nodes can convince the verifier.

In fact, a similar idea can be used to certify that $v \notin C_i$, given $|C_{i-1}|$. The only difference is that in step (3) the verifier checks that neither v, nor any neighbour of v, appears in the list of nodes in C_{i-1}. This procedure works because $v \in C_i$ if and only if there exists $u \in C_{i-1}$ such that either $u = v$ or $u \to v$.

2. We have described certificates for each node v to prove that either $v \in C_i$ or $v \notin C_i$. The certificate that $|C_i| = c$ is just the list of these certificates for each node v in ascending order. The verifier simply checks all these certificates and accepts if and only if the number of nodes certified to be in C_i is equal to c.

□

Self Test Question:
Assuming the Immerman-Szelepcsényi Theorem, prove that 2-SAT $\in$ NL.

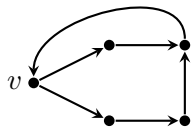
Solution:
By the Immerman-Szelepcsényi Theorem, it suffices to show that determining whether a 2-SAT formula ϕ is *not* satisfiable is in NL. The reduction from 2-SAT to graph reachability described in Theorem 6.2 is implicitly logspace-computable. Following this reduction, unsatisfiability of ϕ can be certified by a path from a variable to its negation, and back again. This path can be checked with read-once access in logarithmic space.

Self Test Question:
Prove that $\mathsf{P} \neq \mathsf{SPACE}(n)$. [Hint: you need not show that either class contains the other.]

Solution:
For any language $\mathcal{L}$, set $\mathcal{L}' = \{(x, 0^{|x|^2}) : x \in \mathcal{L}\}$. First observe that, if $\mathcal{L}' \in \mathsf{P}$, $\mathcal{L} \in \mathsf{P}$. For any $\mathcal{L} \in \mathsf{SPACE}(n^2)$, $\mathcal{L}' \in \mathsf{SPACE}(n)$. Thus, if $\mathsf{P} = \mathsf{SPACE}(n)$, $\mathcal{L}' \in \mathsf{P}$, so $\mathcal{L} \in \mathsf{P}$ and hence $\mathcal{L} \in \mathsf{SPACE}(n)$. But, as this holds for any $\mathcal{L} \in \mathsf{SPACE}(n^2)$, it contradicts the Space Hierarchy Theorem.

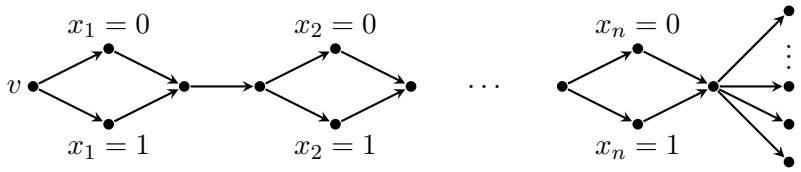
4.5 Advanced Technical Notes

Self Test Question:
($\star$) For any directed graph G, consider the following two-player game. Starting at a given node v and with player 1, players take it in turns to pick an arc from the current node to a node which has not yet been visited. If there is no arc from the current node to an unvisited node, the current player loses. For example, on the following graph, starting with node v, player 1 can always win.



Show that the following problem is PSPACE-complete: given a graph G and node v, determine whether player 1 can win when starting from node v. [HARD]

Solution:
To show that this problem is in PSPACE, we give an explicit polynomial-space algorithm $A(i, G, v)$ which returns 1 if player i wins on graph G, starting from node v, and 0 otherwise. We have $A(1, G, v) = 1$ if and only if there exists a node w such that $v \to w$ and $A(2, G', w) = 0$, where G' is G with v removed. Similarly, $A(2, G', w) = 0$ if and only if, for all u such that $w \to u$, $A(1, G'', u) = 1$, where G'' is G' with w removed. Thus A can be implemented in polynomial space using the same idea as the polynomial-space algorithm for TQBF.
This is not a coincidence, as we can show that the present problem is in fact PSPACE-complete using a reduction from TQBF. Given a quantified boolean formula ψ on an even number n of variables $x_1, \ldots, x_n$ based on a boolean formula ϕ in CNF with m clauses, we construct a pair (G, v) such that player 1 wins starting from v if and only if ψ is true. The graph is formed of n diamonds, each corresponding to a variable and connected in sequence with an arc, finishing with m nodes, each corresponding to a clause.



From each node corresponding to a clause containing k variables there are k arcs (not shown in the graph). If variable x_i appears negated in the clause, there is an arc to the node "$x_i = 0$"; otherwise, there is an arc to the node "$x_i = 1$". Now any path from the start node v can be seen as picking truth assignments for the variables; the nodes visited correspond to negations of the assignments. Observe that player 1 picks the assignments to the odd-numbered variables, and player 2 picks the assignments to the even-numbered variables. Finally, player 2 picks a node corresponding to a clause. If any of the nodes corresponding to literals in that clause have not been visited, player 1 can pick an arc to such a node and wins, as player 2 has nowhere to go on the next move; otherwise, player 1 loses.
Thus player 1 wins if and only if $\exists x_1 \forall x_2 \exists x_3 \ldots \forall x_n \phi(x_1, \ldots, x_n)$, i.e. player 1 wins if and only if ψ is true. This game is usually known as "Generalized Geography".

Lecture 5

Randommized Algorithms

The goal of the Turing machine model introduced in Section 0.4 is to encompass the notion of computation in the physical world. However, there is an aspect of the physical world which this model does not seem to capture: randomness. While it is a philosophical question whether true randomness "really" exists in the universe, for practical purposes it does appear that there exist devices (e.g. coins) which can be used to generate apparently random numbers, which could be used in computation. It is less clear whether such randomness is actually useful, so we begin by discussing two examples of non-trivial randomised algorithms. Before proceeding thought we need to issue a big warning which trips up many a student:

WARNING: Random probabilistic choice is different from non-determinism! (even though both relate a current configuration to many possibilities in a single transition) – in a NDTM the computer makes many steps "simultaneously" following *all* paths in a computational tree. In the probabilistic case the computer chooses only *one* step at each node (probabilistically) and follows only some *one* path through the tree. However on repeating the computation with the same input, the path chosen will generally be a different one. We require the final answer to be correct with "suitably high probability" (cf later for formal definitions of classes and various specific requirements that we impose).

5.1 Some Inequalities Involving Probabilities

When argueing about randomized algorithms we clearly need to discuss facts about probabilities and expectations. The following two Theorems should be readily to hand for any computer scientist. We will assume them as given, but for those interested we give the proofs here. Recall the following basic facts: Linearity of expectation ($\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$) and the union bound $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$.

Theorem 37 (Markov's inequality)**.** *Let X be a random variable. Then, for any $c > 0$,*

$$\Pr[|X| \geq c] \leq \frac{\mathbb{E}[|X|]}{c}.$$

Proof. Let $p_x = \Pr[|X| = x]$. Then, for any $c > 0$,

$$\Pr[|X| \geq c] = \sum_{x \geq c} p_x \leq \frac{1}{c} \sum_{x \geq c} x p_x \leq \frac{1}{c} \sum_x x p_x = \frac{\mathbb{E}[|X|]}{c}.$$

□

A Bernoulli random variable is a random variable which takes the value 1 with probability p, for some $0 \leq p \leq 1$, otherwise taking the value 0. The sum of n independent Bernoulli random variables is tightly concentrated around its mean. This idea can be made precise using Chernoff bounds (also known as Chernoff-Hoeffding bounds or Hoeffding bounds), which are a fundamental tool in the analysis of randomised algorithms.

Theorem 38 (Chernoff bound)**.** *Let $X = \sum_i X_i$, where each X_i is an independent Bernoulli random variable taking the value 1 with probability p_i. Write $\mu = \mathbb{E}[X] = \sum_i p_i$. Then, for all $\delta > 0$,*

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^\mu, \text{ and}$$
$$\Pr[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}} \right)^\mu.$$

Proof. The basic idea is to apply Markov's inequality to the random variable e^{tX}, for some t, and then to optimise over t. For the first part, we have the following chain of equalities and inequalities.

$$\begin{aligned}
\Pr\left[\sum_i X_i \geq (1+\delta)\mu\right] &= \Pr\left[e^{t\sum_i X_i} \geq e^{t(1+\delta)\mu}\right] && \text{[for any } t > 0\text{]} \\
&\leq \frac{\mathbb{E}\left[e^{t\sum_i X_i}\right]}{e^{t(1+\delta)\mu}} && \text{[Markov's inequality]} \\
&= \frac{\prod_i \mathbb{E}[e^{tX_i}]}{e^{t(1+\delta)\mu}} && \text{[by independence of } X_i\text{'s]} \\
&= \frac{\prod_i (1 + p_i(e^t - 1))}{e^{t(1+\delta)\mu}} && \text{[by definition of } X_i\text{'s]} \\
&\leq \frac{e^{(e^t-1)\sum_i p_i}}{e^{t(1+\delta)\mu}} && \text{[using the inequality } 1 + x \leq e^x\text{]} \\
&= e^{\mu((e^t-1)-t(1+\delta))}.
\end{aligned}$$

We now minimise this expression over t. The minimum is obtained at $t = \ln(1+\delta)$, which gives

$$\Pr\left[\sum_i X_i \geq (1+\delta)\mu\right] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^\mu$$

as claimed. For the second part, we follow exactly the same procedure but start with

$$\Pr\left[\sum_i X_i \leq (1-\delta)\mu\right] = \Pr\left[-\sum_i X_i \geq -(1-\delta)\mu\right].$$

□

One can simplify these bounds to obtain the following corollary.

Corollary 39. *Let $X = \sum_i X_i$, where each X_i is an independent Bernoulli random variable taking the value 1 with probability p_i. Write $\mu = \mathbb{E}[X] = \sum_i p_i$. Then, for all $0 \leq \delta \leq 1$,*

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}, \text{ and}$$
$$\Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}.$$

Proof. For the first part, we examine the Taylor expansion of $\ln(1+\delta)$ to obtain

$$(1+\delta)\ln(1+\delta) = \delta + \frac{\delta^2}{2} - \frac{\delta^3}{6} + \frac{\delta^4}{12} - \cdots \geq \delta + \frac{\delta^2}{2} - \frac{\delta^3}{6} \geq \delta + \frac{\delta^2}{3},$$

valid for $0 \leq \delta \leq 1$, which implies $(1+\delta)^{1+\delta} \geq e^{\delta+\delta^2/3}$. For the second part, we bound $(1-\delta)^{1-\delta} \geq e^{-\delta+\delta^2/2}$, which similarly follows from the Taylor expansion of $\ln(1-\delta)$. □

Finally, an additive form of these bounds will often be useful, which we state as the following immediate corollary (with suboptimal constants).

Corollary 40. *Let $\overline{X} = \frac{1}{n}\sum_{i=1}^n X_i$, where each X_i is an independent Bernoulli random variable taking the value 1 with probability p_i. Write $\mu = \mathbb{E}[\overline{X}] = \frac{1}{n}\sum_{i=1}^n p_i$. Then, for all $0 \leq \delta \leq 1$,*

$$\Pr[\overline{X} \geq \mu + \delta] \leq e^{-n\delta^2/3}, \text{ and}$$
$$\Pr[\overline{X} \leq \mu - \delta] \leq e^{-n\delta^2/2}.$$

Proof. Immediate from Corollary 39, observing that $0 \leq \mu \leq 1$. □

Observe that the phrasing of Corollary 40 highlights the fact that, as n increases, $\overline{X}$ becomes rapidly more concentrated around μ.

5.2 Adding randomness to Turing machines

In Section 5.6 below we give some classic examples of where randomness helps us to achieve better results (in practice and in theory) than we would otherwise have. A very simple example of this which you have already seen is Quicksort; if we apply *deterministic* Quicksort to a list of n elements then the *expected* run time is $\mathcal{O}(n \cdot \log n)$ when we take the expectation over all possible input lists, however the worst case run time is $\mathcal{O}(n^2)$. If we pick the pivots

at random in Quicksort, i.e. we think of Quicksort as a *randomized* algorithm, then the run time is still the same but the expectation is independent of the input[1]. An alternative strategy, which obtains the same outcome, is to randomize the input (i.e. shuffle the input before we sort it), and then apply deterministic Quicksort. This last strategy is what is called a *randomized reduction* from the worst case to the average case of sorting.

Hence we see randomize is very useful, and so our equating of "efficient" with *deterministic* polynomial time Turing Machines is far too simplistic.

We now formalise the ability to use randomness in algorithms via the model of the probabilistic Turing machine (PTM). A PTM is a Turing machine M with two transition functions δ_0, δ_1. At each step of its execution, with probability $1/2$ M applies δ_0, and with probability $1/2$ it applies δ_1. The machine only has two possible outputs: 1 (accept) or 0 (reject). For some function $T : \mathbb{N} \to \mathbb{N}$, we say that M halts in $T(n)$ time if, for any input x, M halts within $T(|x|)$ steps (whatever random choices it makes). Let $M(x)$ be the random variable denoting the output of M on input x. See Figure 5.1 for an example execution of a mythical Turing machine on a specific problem.

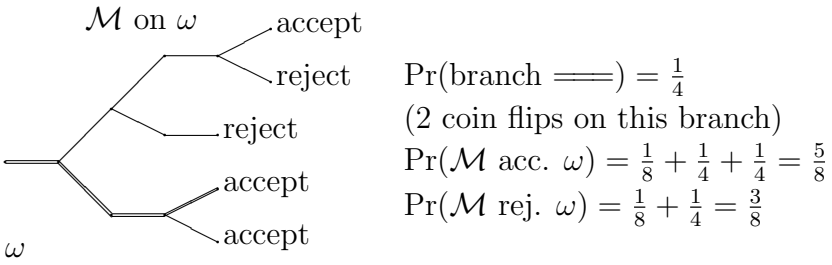


Figure 5.1: Simple example of a randomized Turing machines execution

We can now define the complexity class BPP, which encapsulates our idea of efficient probabilistic computation. For $\mathcal{L} \subseteq \{0,1\}^*$, we say that M decides $\mathcal{L}$ in time $T(n)$ if, for every $x \in \{0,1\}^*$, M halts in $T(|x|)$ steps (regardless of its random choices), and:

- For all $x \in \mathcal{L}$, $\Pr[M(x) = 1] \geq 2/3$;
- For all $x \notin \mathcal{L}$, $\Pr[M(x) = 1] \leq 1/3$.

Let $\mathsf{BPTIME}(T(n))$ be the class of languages decided by some PTM in time $O(T(n))$ and write

$$\mathsf{BPP} = \bigcup_{c>0} \mathsf{BPTIME}(n^c).$$

Observe that an equivalent alternative definition of PTMs would be to take a standard deterministic Turing machine M, and give M an additional "randomness" tape containing random bits.

[1]So if we run the algorithm on the same input many times then we will get an average run time of $\mathcal{O}(n \cdot \log n)$

Algorithms which operate in this way are often called *Two Sided Monte Carlo* algorithms. They are gauranteed to terminate, but they can give an incorrect answer: If $x \in \mathcal{L}$ then $\Pr[M(x) = 1] \geq 2/3$ implies that they could output zero when they should output one, where as if $x \notin \mathcal{L}$ then $\Pr[M(x) = 1] \leq 1/3$ implies they could output one when they should output zero. However, since $2/3 \geq 1/2 \geq 1/3$ then they do get the answer right with a better than average probability (since on average they could just flip a coin and output zero or one).

The constants $2/3$, $1/3$ appearing in this definition may appear somewhat arbitrary. We now show that, in fact, they can be replaced with any pair of numbers either side of $1/2$ separated by at least an inverse polynomial gap.

Lemma 41. *Assume there exists a PTM M that decides $\mathcal{L} \subseteq \{0,1\}^*$ with success probability $1/2 + \gamma$ using time $T(n)$, i.e.:*

- *for all $x \in \mathcal{L}$, $\Pr[M(x) = 1] \geq 1/2 + \gamma$, and*
- *for all $x \notin \mathcal{L}$, $\Pr[M(x) = 1] \leq 1/2 - \gamma$.*

Then $\mathcal{L} \in \mathsf{BPTIME}(T(n)/\gamma^2)$.

Proof. We will boost the success probability of M using majority voting. Our algorithm is simply to run M K times (for some K to be determined), obtaining outcomes $y_1, \ldots, y_K$. If $|\{i : y_i = 1\}| \geq K/2$, we output 1, otherwise we output 0.

Assume $x \notin \mathcal{L}$ (the case $x \in \mathcal{L}$ is similar). Then each $y_i \in \{0,1\}$ is an independent random variable with $\mathbb{E}[y_i] \leq 1/2 - \gamma$. Our algorithm will fail if $\overline{Y} := \frac{1}{K}\sum_{i=1}^{K} y_i \geq 1/2$. We have $\mathbb{E}[\overline{Y}] \leq 1/2 - \gamma$. By the Chernoff bound (Corollary 40),

$$\Pr[\overline{Y} \geq 1/2] \leq e^{-\gamma^2 K/3},$$

so taking $K = O(1/\gamma^2)$ suffices to upper bound the probability of failure by $1/3$. Observe that, if we increase K further than this, the success probability goes to 1 exponentially fast! □

As we see from this proof, given a BPP algorithm for deciding some language $\mathcal{L}$, we can write down another BPP algorithm which decides $\mathcal{L}$ with *exponentially small* worst-case failure probability. One may wonder whether it is possible to relax the constraints on the probabilities further without changing the definition of BPP. Let PP be the class of languages $\mathcal{L}$ decided by a polynomial-time PTM such that

- For all $x \in \mathcal{L}$, $\Pr[M(x) = 1] > 1/2$;
- For all $x \notin \mathcal{L}$, $\Pr[M(x) = 1] \leq 1/2$;

Then we have the following result.

Theorem 42. $\mathsf{NP} \subseteq \mathsf{PP}$.

Proof. Suppose that $\mathcal{L} \in \mathsf{NP}$ and let M be a polynomial-time NDTM with two transition functions (i.e. at each computational step, two possibilities for what to do next). We can therefore treat M as a PTM by flipping a coin at each step to decide which transition function to use. Consider a machine M' which has an additional step at the beginning: with probability $1/2$, run M; and with probability $1/2$, accept. If the input $x \in \mathcal{L}$, then M has at least one accepting path on x, so $\Pr[M'(x) = 1] > 1/2$. If $x \notin \mathcal{L}$, then M has no accepting paths, so $\Pr[M'(x) = 1] = 1/2$. Thus $\mathcal{L} \in \mathsf{PP}$. □

It therefore seems unlikely that $\mathsf{BPP} = \mathsf{PP}$. Indeed, many conjecture that in fact $\mathsf{BPP} = \mathsf{P}$. This would accord with our intuition that, in order to simulate "truly random" numbers, it suffices to produce suitably good "pseudorandom" numbers deterministically. NP and BPP are essentially two classes of poly time algorithms, each with a further extra "resource": non-determinism for NP and probabilistic choice fop BPP. It is not known how to compare these resources i.e. we do not know if NP⊆BPP or BPP⊆NP or (more likely?) if they are incomparable: All we know is the inclusions in Figure 5.2.

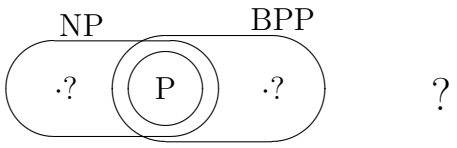


Figure 5.2: BPP vs P vs NP

The question of whether $\mathsf{BPP} = \mathsf{P}$, and "derandomisation" questions more generally, are discussed at length in Arora-Barak chapters 20 and 21. However, it is currently not even known whether $\mathsf{BPP} \subseteq \mathsf{NP}$ (although it *is* known that $\mathsf{BPP} \subseteq \Sigma_2!$).

Self Test Question:
Prove that $\mathsf{PP} \subseteq \mathsf{PSPACE}$.

Solution:
Given an NP machine, we can count the number of accepting paths exactly in polynomial space, by trying each of the $2^{\mathrm{poly}(n)}$ paths in turn, checking whether each path accepts, and incrementing a counter if it does. Hence we can determine if the number of accepting paths is greater than or less than $1/2$ of the total number of paths.

Self Test Question:
Imagine we generalise the definition of BPP to a class $\mathsf{BPP}_{\epsilon,\delta}$ defined as follows. Letting M be a PTM as in the standard definition, we say that $\mathcal{L} \in \mathsf{BPP}_{\epsilon,\delta}$ if there exists an M such that:

- For all $x \in \mathcal{L}$, $\Pr[M(x) = 0] \leq \epsilon$;
- For all $x \notin \mathcal{L}$, $\Pr[M(x) = 1] \leq \delta$.

Thus $\mathsf{BPP} = \mathsf{BPP}_{\frac{1}{3},\frac{1}{3}}$.

1. What are the classes $\mathsf{BPP}_{0,0}$ and $\mathsf{BPP}_{\frac{1}{2},\frac{1}{2}}$?
2. Show that $\mathsf{BPP}_{\frac{1}{2},0} \subseteq \mathsf{NP}$.
3. Show that $\mathsf{BPP}_{2^{-|x|},2^{-|x|}} = \mathsf{BPP}$.
4. What is the class $\mathsf{BPP}_{2^{-2^{|x|}},2^{-2^{|x|}}}$?

Solution:

1. $\mathsf{BPP}_{0,0} = \mathsf{P}$ as the algorithm has to succeed with certainty. $\mathsf{BPP}_{\frac{1}{2},\frac{1}{2}}$ is the class of all languages, as the machine which accepts with probability $1/2$ on all inputs fulfils the definition.
2. NP is the class of languages $\mathcal{L}$ such that there is an NDTM which has at least one accepting path for $x \in \mathcal{L}$, and no accepting paths for $x \notin \mathcal{L}$. $\mathsf{BPP}_{\frac{1}{2},0}$ is the class of languages $\mathcal{L}$ such that there is an NDTM which has at least a $1/2$ fraction of accepting paths for $x \in \mathcal{L}$, and no accepting paths for $x \notin \mathcal{L}$. Thus the latter is a subset of the former.
3. The proof of Lemma 9.8 implies that by repeating a BPP computation polynomially many times, we can achieve exponentially small failure probability.
4. $\mathsf{BPP}_{2^{-2^{|x|}},2^{-2^{|x|}}} = \mathsf{P}$. A BPP algorithm can only make $\mathrm{poly}(|x|)$ random choices. Thus, if $\Pr[M(x) = 0] < 2^{-\mathrm{poly}(|x|)}$, in fact $\Pr[M(x) = 0] = 0$ (and similarly for $\Pr[M(x) = 1]$).

Self Test Question:
Imagine we generalise the definition of BPP by allowing the PTM to run in *expected* time polynomial in the input size. That is, the expected running time of the PTM is polynomial on every input (but sometimes, depending on its internal randomness, it might run for much longer). Does this change the definition of BPP?

Solution:
No. Given such a PTM M which runs in expected time T, consider the PTM M' which runs M for $10T$ steps (say), and outputs the result of M (if M has halted) or a random answer (if M has not halted). By Markov's inequality, M will have halted within this time with probability at least $9/10$. If M halts, it gives the right answer with probability at least $2/3$. If M has not halted, we get the right answer with probability $1/2$. Thus the probability that M' gives the right answer is $9/10 \times 2/3 + 1/10 \times 1/2 = 13/20$; repeating this process and taking a majority vote gives an algorithm with success probability at least $2/3$, as required.

5.3 Complexity Classes: The Summary

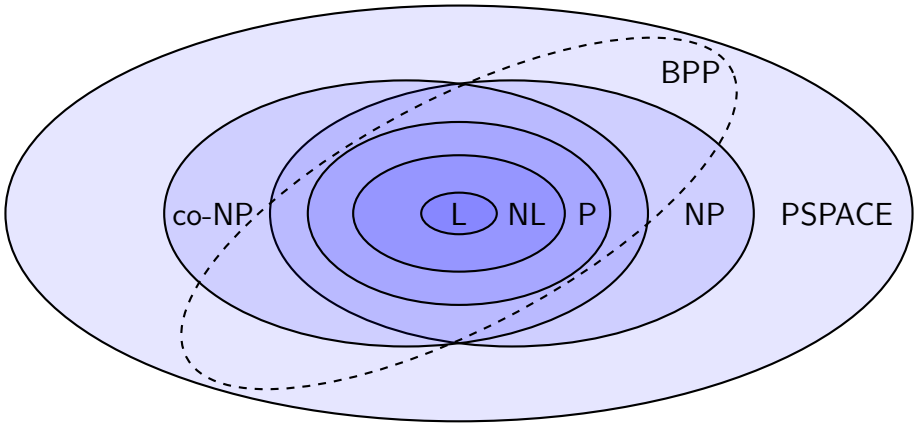


Figure 5.3: Some important complexity classes.

Figure 5.3 summarises known relationships between many of the complexity classes we have met so far. All of these inclusions (with the possible exception of $\mathsf{P} \subseteq \mathsf{BPP}$) are believed to be strict, but there is no proof that this is the case. Observe that we have the chain of inclusions

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}.$$

By the Space Hierarchy Theorem, we know that $\mathsf{L} \neq \mathsf{PSPACE}$. Therefore, at least one of these inclusions must be strict – but we do not know which one.

The class BPP is often equated with "feasible computations", since randomness often makes algorithms go much faster so allowing it seems a good idea in practice. There is a great deal of work in trying to prove that $\mathsf{P} = \mathsf{BPP}$, via a process called de-randomization. Basically taking a randomized algorithm and then removing the random numbers required by the algorithm. The process is to try to reduce the number of real random numbers to as small as zero, if they get to zero for all algorithms in BPP then we would have $\mathsf{BPP} = \mathsf{P}$. This process is also important in practice as true random numbers are hard to come by, so most probabilistic algorithms utilize a pseudo-random number generator (PRNG). A PRNG generates fake random numbers from a small source of entropy (i.e. real random numbers).

5.4 The Class RP: One Sided Monte Carlo Algorithms

The class BPP define the set of two sided Monte Carlo algorithms. If we want to ensure that "true" is never output by mistake we have what are called one sided Monte Carlo algorithms, and these are captured by the complexity class RP.

A language $\mathcal{L}$ is in RP if there is a poly time PTM $\mathcal{M}$ with:

- For all $x \in \mathcal{L}$, $\Pr[M(x) = 1] \geq 1/2$;
- For all $x \notin \mathcal{L}$, $\Pr[M(x) = 0] = 1$.

Hence there can be no false "accepting" answers. If M accepts x then certainly $x \in L$. But there can be false rejections: if M rejects x then it can still be possible that $x \in \mathcal{L}$. However the occurrence of such false rejections is limited to probability at most half. In this sense RP is said to have *one-sided error* in contrast to BPP where both answers (accept or reject) have (suitably small but nonzero) probabilities of error. We say that BPP has *two-sided error*.

co-RP is the class of languages that are the complement of some language in RP. Thus if $\mathcal{L}' \in$ co-RP then there is a poly time PTM M (interchanging the roles of accept and reject in M above) such that

- For all $x \in \mathcal{L}'$, $\Pr[M(x) = 1] = 1$;
- For all $x \notin \mathcal{L}'$, $\Pr[M(x) = 0] \geq 1/2$.

For co-RP there are no false rejections but there are false accepting outputs. It is not known whether RP = co-RP or not, but we do have

Lemma 43. RP $\subseteq$ BPP.

Proof. Suppose $\mathcal{L} \in$ RP and M is a poly time PTM with

- If $w \in \mathcal{L}$ then $\Pr[M(w) = 1] \geq 1/2$;
- if $w \notin \mathcal{L}$ then $\Pr[M(w) = 0] = 1$.

Then M does not itself show that $\mathcal{L} \in$ BPP (because the probability in the first conditation above is only $\geq 1/2$, not $\geq 2/3$) but let M^* be the machine: run M twice. If at least one outcome is 'accept' then accept. If both 'reject' then reject. Then if $w \in \mathcal{L}$ we obtain $\Pr[M^*(w) = 0] \leq 1/4$ so $\Pr[M^*(w) = 1] \geq 3/4 > 2/3$. If $w \notin \mathcal{L}$ then $\Pr[M^*(w) = 0] = 1 > 2/3$. Hence M^* shows that $\mathcal{L} \in$ BPP. □

5.5 The Class ZPP: Las Vegas Algorithms

Technically we define ZPP = RP $\cap$ co-RP. But there are many equivalent definitions for example $\mathcal{L} \in$ ZPP if there is a PTM M such that

- For all $x \in \mathcal{L}$, $\Pr[M(x) = 1] = 1$.

- For all $x \notin \mathcal{L}$, $\Pr[M(x) = 0] = 1$.
- The running time of $M(x)$ is *expected* to be polynomial for every input x.

i.e. It will always return the correct answer, but whilst the average running time is polynomial it might occasionally run for much longer (depending on what random choices the machine M makes).

Another definition is $\mathcal{L} \in \mathsf{ZPP}$ if there is a PTM M' such that

- For all x, $M'(x)$ runs in polynomial time.
- The machine can output 0, 1 or ?.
- For all $x \in \mathcal{L}$, $\Pr[M'(x) = 0] = 0$.
- For all $x \notin \mathcal{L}$, $\Pr[M'(x) = 1] = 0$.
- For all x, $\Pr[M'(x) =?] \leq 1/2$.

Thus it either gives the correct answer, or returns ?. These two definitions are equivalent: If we have an M' we can create an M by running M' multiple times until it gives us a correct answer. If on the other hand we have an M we let T denote the expected running time of M. We then create an M' by running M, terminating after $2 \cdot T$ steps. Thus the run time is $2 \cdot T$, i.e. polynomial. If the instances give an answers then output this answer, otherwise output ?. The probability that it does not give an answer is, by Markov's inequality, less than one half. Thus our M' satisfies the given definition.

5.6 Supplementary Notes

Here we give some examples of randomized algorithms. The first is a one side Monte-Carlo algorithm for checking polynomial identities. The second is the most "efficient" (in practice) test for primality; although a polynomial time deterministic test is known, the randomized algorithm is better. The third is a randomized algorithm for k-SAT. The fourth is a Las Vegas algorithm for finding the factors of $N = p \cdot q$ when given e and d such that $e \cdot d = 1 \pmod{\Phi(N)}$. This algorithm has cryptographic applications, but is presented here as it is the only simple Las Vegas algorithm I could find.

5.6.1 Polynomial Identities

Suppose we are given a polynomial $Q(x_1, \ldots, x_n)$ in n variables (with integer coefficients, say) and we want to know if it is identically zero or not. For example if we did not know Vandermonde's identity we might want to check the truth of

$$Q = \det \begin{vmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & \ldots & & x_2^{n-1} \\ \vdots & & & & \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{vmatrix} - \prod_{j<i} (x_i - x_j) = 0$$

for $n = 100$ say. (Vandermonde's identity states that Q is in fact identically zero, for any choices of the variables). Direct expansion and algebraic simplification is hopeless – an $n \times n$ determinant (with algebraic entries) generally has $n!$ terms and $n!$ grows superexponentially with n (but determinants with *numerical* entries can be evaluated in poly time using Gaussian elimination). Instead we'll give a "randomised" algorithm based on the following result:

Lemma Let $Q(x_1, \ldots, x_n)$ be a polynomial of degree at most d in each of the n variables $x_1, \ldots, x_n$. Suppose Q is not identically zero. If I (with $|I|$ elements) is any set of trial x values such that $|I| \geq cd$ (for some constant c) then the number of n-tuples $(x_1, \ldots, x_n) \in I \times \ldots \times I$ which are zeroes of Q, is at most $n|I|^n/c$ (i.e. a fraction n/c of all possibilities).

Proof We use proof by induction on the number of variables n. For $n = 1$, $Q(x_1)$ is a poly of degree at most d and the number of roots is known to be at most d, which is $\leq |I|/c$ by the choice of $|I|$. Now suppose

$$\text{the lemma is true for } n = k - 1. \tag{5.1}$$

Based on this supposition we'll prove it's true for $n = k$. For $n = k$ write Q as

$$Q = p_0(x_1, \ldots, x_{k-1}) + p_1(x_1, \ldots, x_{k-1})x_k + \ldots + p_d(x_1, \ldots, x_{k-1})x_k^d$$

i.e. a poly in x_k with coefficients that are polys in $x_1, \ldots, x_{k-1}$. Suppose we have a zero $(x_1, \ldots, x_k)$ of Q. Then with these x values p_d is either (a) zero or (b) not zero. For the case (a) there are $(k-1)|I|^{k-1}/c$ possibilities for $(x_1, \ldots, x_{k-1})$ (using our assumption (5.1)) and $|I|$ possibilities for x_k i.e. at most $(k-1)|I|^k/c$ possibilities in all. In case (b) Q is a poly of degree d in x_k having at most d roots for each of the $|I|^{k-1}$ choices of $(x_1, \ldots, x_{k-1})$. But $d \leq |I|/c$ so we get at most $|I|^k/c$ roots. Adding the possibilities for cases (a) and (b) we see that Q can have at most $k|I|^k/c$ roots, proving the lemma for $n = k$ on the assumption that it is true for $n = k - 1$. Hence by induction, the lemma is true for all n. □

Randomised algorithm to check truth of the claim $Q(x_1, \ldots, x_n) = 0$:
Choose I so that $|I| > 2n\deg(Q)$ (i.e. take $c = 2n$ in the lemma). Hence if we choose $(x_1, \ldots, x_n)$ at random from $I \times \ldots \times I$ we'll have probability $\leq 1/2$ of hitting a root, if Q is not identically zero.
So choose N such n-tuples independently at random and evaluate Q on them. If Q is not identically zero then the probability that they all give zero is $\leq 1/2^N$. Thus take N large enough so that $1/2^N$ is sufficiently small for our purposes e.g. $N = 1000$, and evaluate Q N times. If we get all zeroes then output "Q is zero". If we get at least one non-zero value, output "Q is not zero". The probability of making an error is then $\leq 1/2^N = 1/2^{1000}$. Note also that this error is *one-sided*: if we output "Q is not zero" then this is *always* correct (since we have seen a non-zero value of Q) but the output "Q is zero" is sometimes wrong (but with very small probability).

5.6.2 Primality testing

Imagine that we are given an n-digit integer N, and would like to determine whether N is prime. The input size is n, so we would like an algorithm that runs in time poly(n), i.e. poly($\log N$). In particular, note that the naïve algorithm which tests all potential divisors of N (from 2 up to $\lfloor\sqrt{N}\rfloor$) does not run in time poly(n). We now give a polynomial-time randomised test for primality, which is known as the Miller-Rabin test.

1. Assume that N is odd, and write $N-1$ as $2^j s$ for some odd s and integer j.
2. Pick a random integer $r \in \{1, \ldots, N-1\}$.
3. Compute the sequence $S = r^s, r^{2s}, r^{4s}, \ldots, r^{2^j s}$ modulo N by repeated squaring.
4. Output "composite" if either:
   (a) $r^{2^j s} \not\equiv 1$, or
   (b) for some i, the sequence goes from $r^{2^i s} \notin \{1, N-1\}$ to $r^{2^{i+1} s} \equiv 1$.
5. Otherwise, output "prime".

Theorem 44. *If N is prime, the Miller-Rabin test always returns "prime". If N is composite, the Miller-Rabin test returns "composite" with probability at least $1/2$.*

For any n, let $\mathbb{Z}_n$ denote the additive group of integers modulo n, and let $\mathbb{Z}_n^\times$ denote the multiplicative group of integers modulo n, i.e. the group consisting of integers a such that $0 < a < n$ and a is coprime to n, under multiplication. Recall that a is coprime to n if and only if there exists an integer b such that $ab \equiv 1 \pmod{n}$. In order to prove Theorem 44, we will need the following two facts from basic number theory, which we state without proof.

- Fermat's little theorem: For any prime p, and any positive integer $a < p$, $a^{p-1} \equiv 1 \pmod{p}$.
- The Chinese remainder theorem: For a positive integer N with prime factorisation $N = p_1^{r_1} \ldots p_k^{r_k}$, $\mathbb{Z}_N^\times \cong \mathbb{Z}_{p_1^{r_1}}^\times \times \cdots \times \mathbb{Z}_{p_k^{r_k}}^\times$.

Proof of Theorem 44. First, if N is prime, then by Fermat's little theorem $r^{2^j s} \equiv 1 \pmod{N}$, so the last element of S is 1. Therefore, the sequence is either all 1's, or becomes 1 following a squaring operation. But in the field $\mathbb{Z}_N$ the only roots of the equation $x^2 = 1$ are $x = \pm 1$, so the test always outputs "prime" when N is prime. We now use some group theory to prove that the test works with probability at least $1/2$ when N is composite.

First assume that there exists an $a \in \mathbb{Z}_N^\times$ such that $a^{2^j s} \not\equiv 1$ (i.e. via Fermat's little theorem, a is a "witness" that N is composite). The a's such that $a^{2^j s} \equiv 1 \pmod{N}$ form a subgroup of $\mathbb{Z}_N^\times$. As this subgroup is proper, it contains at most half of the elements of $\mathbb{Z}_N^\times$. So, when r is picked at random, it is either coprime to N (in which case with probability at least $1/2$, $r^{2^j s} \not\equiv 1$), or it is not coprime to N (in which case $r^{2^j s} \not\equiv 1$ always).

What if this does not work – i.e. for all $a \in \mathbb{Z}_N^\times$, $a^{2^j s} \equiv 1$? Such N exist and are called *Carmichael numbers* (the smallest is 561). In this case, we show that the algorithm still works. Consider the map $r \mapsto r^s$ which we perform first. This is a homomorphism from the group $\mathbb{Z}_N^\times$ to some subgroup H_1 (i.e. $(ab)^s = a^s b^s$ for all $a, b \in \mathbb{Z}_N^\times$), so r^s is uniformly distributed in H_1. We then repeatedly apply the map $a \mapsto a^2$, giving a sequence of subgroups $H_2, \ldots, H_{j+1}$, where $H_{j+1} = \{1\}$ because N is a Carmichael number. This implies that the order of every element of each subgroup H_i is a power of 2, so $|H_i|$ is a power of 2.

Now consider the subgroup I which is the last subgroup before we get to $\{1\}$. The numbers r that pass the Miller-Rabin test must have been mapped to either 1 or -1 at this stage. I may or may not contain the element -1. If it does not contain -1, as $|I| \geq 2$, r fails the Miller-Rabin

test with probability at least 1/2. If it contains -1 and at least one more element (other than 1), $|I| \geq 4$ (as $|I|$ is a power of 2), so the same conclusion follows.

We now show that if $-1 \in I$, so is another element (other than 1). If N is not a power of a prime, by the Chinese remainder theorem $\mathbb{Z}_N^\times$ is isomorphic to a product $J \times K$ of nontrivial groups. So the element $-1 \in \mathbb{Z}_N^\times$ can be represented as a pair $(-1_J, -1_K)$. Consider the elements $(-1_J, 1_K)$ and $(1_J, -1_K)$. If $-1 \in I$, this is because the map $r \mapsto r^{2^i s}$, for some i, can be applied to some element $(x, y) \in \mathbb{Z}_N^\times$ to obtain $(-1_J, -1_K)$. But then the same map can be applied to $(x, 1)$ and $(1, y)$ to obtain $(-1_J, 1_K)$, $(1_J, -1_K)$, so these two elements are also in I.

We finally claim that, if N is a power of a prime, it is not a Carmichael number, so the test also works in this case. Assume $N = p^k$ and take $a = p + 1$. We show that $a^N \not\equiv a \pmod{p^2}$, which implies that $a^N \not\equiv a \pmod{p^k}$, so $a^{N-1} \not\equiv 1 \pmod{N}$ and hence N is not a Carmichael number. By taking $(p+1)^p = \sum_{\ell=0}^{p} \binom{p}{\ell} p^\ell \equiv 1 \pmod{p^2}$, we have $a^N \equiv 1 \pmod{p^2}$, implying $a^N \not\equiv a$. □

It has been shown quite recently that in fact primality testing is in P. However, the best deterministic algorithm currently known still has a fairly large runtime ($O(n^6 \operatorname{polylog}(n))$), so in practice randomised algorithms are still used for primality testing.

There are many good sets of lecture notes online about randomised algorithms (the discussion here of primality testing is partly based on one such lecture[2]). Papadimitriou chapter 11 gives an extensive analysis of an alternative test for primality by Solovay and Strassen. The deterministic primality test was invented by Agrawal, Kayal and Saxena (two of whom were undergraduates at the time) in 2002.

5.6.3 A randomised algorithm for k-SAT

We now return once more to the much-studied k-SAT problem (recall: given a boolean formula $\phi(x_1, \ldots, x_n)$ in CNF with at most k variables per clause, does there exist an assignment to the variables x_i such that ϕ evaluates to true?). Consider the following algorithm $\mathcal{A}$.

1. Initially assign x_i randomly for all i.
2. Repeat the following t times, for some t to be determined:
   (a) If ϕ is satisfied by x, return that ϕ is satisfiable.
   (b) Otherwise, pick an arbitrary clause C in ϕ which is not satisfied by x. Pick one of the variables x_i in C at random and update x by mapping $x_i \mapsto \neg x_i$.
3. Return that ϕ is not satisfiable.

We will prove the following claims:

Theorem 45. *If $k = 2$ and we take $t = 2n^2$, $\mathcal{A}$ succeeds with probability at least 1/2.*

Theorem 46. *If $k > 2$ and we take $t = 3n$, $\mathcal{A}$ succeeds with probability at least $\left(\frac{k}{2(k-1)}\right)^n / \operatorname{poly}(n)$.*

[2] `http://www.cse.buffalo.edu/~regan/cse681/notes/lectureB12.pdf`

Observe that, for any integer $K > 0$, Theorem 45 implies an algorithm for 2-SAT which fails with probability at most 2^{-K} and uses time $O(Kn^2)$: simply repeat $\mathcal{A}$ K times, return "satisfiable" if any of these uses of $\mathcal{A}$ returned a satisfying assignment, and return "unsatisfiable" otherwise. The probability that $\mathcal{A}$ fails every time is at most 2^{-K}. Similarly, Theorem 46 implies an algorithm for k-SAT which succeeds with probability at least 0.99 (say) in time $O\left(\left(\frac{2(k-1)}{k}\right)^n \text{poly}(n)\right)$. For small k, this can be much better than the trivial algorithm of just trying all possible assignments to x; e.g. for $k = 3$ we get $O((4/3)^n \text{poly}(n))$, as compared with the $\Omega(2^n)$ time taken by the trivial algorithm. This illustrates that even NP-complete problems can have non-trivial algorithms.

The proofs of Theorems 45 and 46 will both be based on the same idea, of reducing the algorithm's behaviour to a random walk. If ϕ is satisfiable, there exists a satisfying assignment x^*. For any $x \in \{0,1\}^n$, let $d(x, x^*)$ be the Hamming distance between x and x^*, i.e. the number of positions at which x and x^* differ. At each step of the algorithm, as a single bit of x is flipped, $d(x, x^*)$ either increases by 1 or decreases by 1. If $d(x, x^*)$ ever becomes 0, the algorithm has found a satisfying assignment to ϕ (i.e. x^*). Importantly, the probability that $d(x, x^*)$ decreases by 1 is at least $1/k$, which we see as follows. Let C be the clause picked by the algorithm, and assume it contains variables $x_{i_1}, \ldots, x_{i_k}$. As x^* satisfies ϕ (and hence C) but x does not satisfy C, x and x^* must differ in at least one position $i \in \{i_1, \ldots, i_k\}$. Hence the probability that i is picked by the algorithm is at least $1/k$. Therefore, the behaviour of the algorithm can be mapped to a random walk on the line indexed by integers $0, \ldots, n$, which moves left with probability at least $1/k$ at each step; see Figure 5.4 for an illustration. We simplify this to a random walk which moves left with probability exactly $1/k$ at each step; calculating the expected number of steps until this walk hits 0 then gives an upper bound on the expected number of steps until $\mathcal{A}$ finds x^*. We have assumed that each clause C contains exactly k variables, but if C contains fewer than k variables this can only increase the probability of moving left.

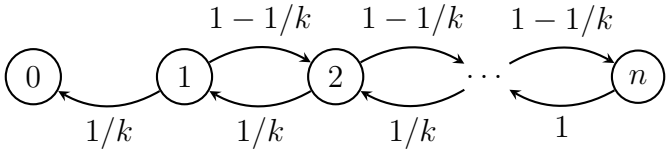


Figure 5.4: The random walk performed by algorithm $\mathcal{A}$ for k-SAT.

We are now ready to prove Theorems 45 and 46.

Proof of Theorem 45. The claim will follow from Markov's inequality by showing that the expected time to hit 0 is at most n^2. This is a well-known result from probability theory, but we give a simple direct argument.

For $0 \leq i \leq n$, let $T(i)$ denote the expected number of steps until the random walk hits 0, given that it started at position i. It should be clear that:

- $T(0) = 0$;
- $T(n) = 1 + T(n-1)$ [the walk cannot move further right than n];
- $T(i) = 1 + \frac{1}{2}(T(i+1) + T(i-1))$ for $1 \leq i \leq n-1$.

This recurrence can readily be solved to yield $T(i) = i(2n - i)$. Thus the expected time to hit 0 is at most n^2 for any initial assignment x. □

Proof of Theorem 46. The proof of this theorem is along the same lines as Theorem 45, but somewhat more complicated.

Imagine the walker starts at position j and walks for T steps. Let P_j be the probability to reach 0 at some time between 1 and T. For any integer ℓ, the probability to go left $j + \ell$ times during the walk is lower bounded by

$$\binom{T}{j+\ell} k^{-(j+\ell)} (1 - 1/k)^{T-(j+\ell)}.$$

Now take $T = (1 + 2\alpha)j$, $\ell = \alpha j$, for some α. The probability to go left $(1 + \alpha)j$ times (and hence right αj times) lower bounds the probability to end up at 0 after at most T steps, giving the bound that

$$P_j \geq \binom{(1+2\alpha)j}{\alpha j} k^{-(1+\alpha)j} (1 - 1/k)^{(1-\alpha)j}.$$

We now choose $\alpha = 1/(k - 2)$. Observe that this implies that $T \leq (1 + 2/(k - 2))n \leq 3n$. Up to poly$(j)$ terms, we have

$$\binom{(1+2\alpha)j}{\alpha j} = \left(\left(\frac{1+2\alpha}{\alpha} \right)^{\alpha} \left(\frac{1+2\alpha}{1+\alpha} \right)^{1+\alpha} \right)^{j},$$

which follows from Stirling's approximation

$$\sqrt{2\pi n} \left(\frac{n}{e} \right)^n e^{1/(12n+1)} \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e} \right)^n e^{1/(12n)},$$

so up to poly(j) terms,

$$P_j \geq \left(\left(\frac{1+2\alpha}{\alpha} \right)^{\alpha} \left(\frac{1+2\alpha}{1+\alpha} \right)^{1+\alpha} \right)^{j} k^{-(1+\alpha)j} (1 - 1/k)^{(1-\alpha)j} = \left(\frac{1}{k-1} \right)^{j}.$$

The start position j is given by the Hamming weight (i.e. the number of 1s) of a random n-bit string x. Thus, using the binomial theorem, up to polynomial terms the probability to reach 0 in $T \leq 3n$ steps is equal to

$$P = \frac{1}{2^n} \sum_{j=0}^{n} \binom{n}{j} P_j \geq \frac{1}{2^n} \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{k-1} \right)^{j} = \left(\frac{k}{2(k-1)} \right)^{n}$$

as claimed. □

The randomised algorithm for k-SAT given here was invented by Schöning in 1999; the case of 2-SAT was already known (Papadimitriou chapter 11 has a discussion). There are several other non-trivial (but still exponential-time!) algorithms known for k-SAT, some of which are deterministic.

5.6.4 Factoring $N = p \cdot q$ given e, d such that $e \cdot d = 1 \pmod{\Phi(N)}$

Recall that for some integer s

$$e \cdot d - 1 = s \cdot (p-1) \cdot (q-1).$$

We pick an integer $x \neq 0$, this is guaranteed to satisfy

$$x^{e \cdot d - 1} = 1 \pmod{N}.$$

We now compute a square root y_1 of one modulo N,

$$y_1 \leftarrow \sqrt{x^{e \cdot d - 1}} = x^{(e \cdot d - 1)/2},$$

which we can do since $e \cdot d - 1$ is known and will be even. We will then have the identity

$${y_1}^2 - 1 \equiv 0 \pmod{N},$$

which we can use to recover a factor of N via computing

$$\gcd(y_1 - 1, N).$$

But this will only work when $y_1 \neq \pm 1 \pmod{N}$.

Now suppose we are unlucky and we obtain $y_1 = \pm 1 \pmod{N}$ rather than a factor of N. If $y_1 = -1 \pmod{N}$, then we set $y_1 \leftarrow -y_1$. Thus we are always left with the case $y_1 = 1 \pmod{N}$. We take another square root of one via,

$$y_2 \leftarrow \sqrt{y_1} = x^{(e \cdot d - 1)/4}.$$

Again we have

$${y_2}^2 - 1 = y_1 - 1 = 0 \pmod{N}.$$

Hence we compute

$$\gcd(y_2 - 1, N)$$

and see if this gives a factor of N. Again this will give a factor of N unless $y_2 = \pm 1$, if we are unlucky we repeat once more and so on.

This method can be repeated until either we have factored N or until $(e \cdot d - 1)/2^t$ is no longer divisible by 2. In this latter case we return to the beginning, choose a new random value of x and start again.

Self Test Question:
Modify the randomised algorithm for k-SAT to give a randomised algorithm which determines with worst-case success probability at least 0.99 whether a graph G with n vertices can be properly 3-coloured. Prove that your algorithm runs in time $O((3/2)^n \operatorname{poly}(n))$, beating the naïve algorithm of exhaustively trying every possible colouring, which takes time $\Omega(3^n)$.

Solution:
We give an algorithm which is a natural variant of the algorithm for k-SAT. A colouring c is an assignment of colours $c_1, \ldots, c_n$ to vertices $v_1, \ldots, v_n$.

1. Initially assign c_i randomly for all i.
2. Repeat the following t times, for some t to be determined:
   (a) If c is a proper 3-colouring of G, return that G is 3-colourable.
   (b) Otherwise, pick an arbitrary pair of vertices v_1, v_2 that are adjacent in G but which are assigned the same colour. Pick $v_i \in \{v_1, v_2\}$ at random and update c by mapping c_i to a random different colour.
3. Return that G is not 3-colourable.

Assume that there is a proper 3-colouring c^* of G and for any colouring c, let $d(c, c^*)$ denote the number of vertices coloured differently to c^*. Let c, c' denote the old and new colourings of G at a given step of the algorithm. If v_1 and v_2 are both incorrectly coloured (i.e. are assigned different colours to their colouring under c^*), with probability $1/2$, $d(c', c^*) = d(c, c^*) - 1$, and with probability $1/2$, $d(c', c^*) = d(c, c^*)$. On the other hand, if v_1 is correctly coloured and v_2 is not:

- With probability $1/4$, $d(c', c^*) = d(c, c^*) - 1$;
- With probability $1/4$, $d(c', c^*) = d(c, c^*)$;
- With probability $1/2$, $d(c', c^*) = d(c, c^*) + 1$.

Call a recolouring $c \mapsto c'$ such that $d(c', c^*) \neq d(c, c^*)$ *useful*. For each useful recolouring, the probability that $d(c', c^*) = d(c, c^*) - 1$ is $1/3$. Looking at the sequence of recolourings, if we only consider useful recolourings, the algorithm can be analysed similarly to the randomised algorithm for 3-SAT.
Let P_j denote the probability that the algorithm finds c^*, given that it starts with a colouring c such that $d(c, c^*) = j$ and makes $T = 3n$ useful recolourings. Following the analysis in the notes, up to poly(j) terms, $P_j \geq 2^{-j}$. Each vertex of the initial random colouring has probability $1/3$ of being coloured the same as in c^*. Thus the probability of finding c^* after T useful recolourings is (up to polynomial terms)

$$\sum_{j=0}^{n} \binom{n}{j} (2/3)^j (1/3)^{n-j} P_j \geq \sum_{j=0}^{n} \binom{n}{j} (2/3)^j (1/3)^{n-j} (1/2)^j \geq 3^{-n} \sum_{j=0}^{n} \binom{n}{j} = (2/3)^n.$$

As with the algorithm for 3-SAT, we need to ensure that the algorithm makes at least $3n$ useful recolourings. Given that the algorithm recolours vertices t times in total, the expected number of useful recolourings is at least $t/2$. Using the Chernoff bound, the probability that the number of useful recolourings is below $t/4$ (say) is at most e^{-Ct} for some universal constant C. Thus it suffices to take $t = O(n \log n)$ to have at least $3n$ useful recolourings with constant probability. So, for this value of t, the algorithm succeeds with probability at least $(2/3)^n$. $(3/2)^n$ repetitions of the whole algorithm thus suffice to find a 3-colouring, if one exists, with failure probability which is an arbitrarily small constant.

5.7 Advanced Technical Notes

Self Test Question:
Give alternative non-trivial randomised and deterministic algorithms for 3-colouring. Possible examples include a different randomised algorithm for 3-colouring which uses time $O((3/2)^n \operatorname{poly}(n))$, and a deterministic algorithm which uses time $O(1.94^n \operatorname{poly}(n))$. [HARD]
[Hints: For the first part, guess which colour should *not* label each vertex. For the second part, at most $n/3$ of the vertices can be labelled with one of the three colours.]

Solution:
For the randomised algorithm, we guess which colour should *not* be placed on each vertex of an n-vertex graph. Our guess will be right with probability $(2/3)^n$. Given a correct guess, we have at most 2 colour choices for each vertex. Which colour is assigned to a given vertex can therefore be expressed as a boolean variable, and the constraint that two adjacent vertices cannot have the same colour can be written as a 2-SAT clause. Thus, determining whether the whole graph is 3-colourable can be achieved in time $\operatorname{poly}(n)$. The expected time taken by the whole algorithm is thus $O((3/2)^n \operatorname{poly}(n))$.
For the deterministic algorithm, we observe that one of the three colours must be used by at most $n/3$ vertices. So we try each of the $f(n) := \sum_{k=0}^{n/3} \binom{n}{k}$ possibilities for assigning a particular colour to at most $n/3$ vertices. In each case such that there is no pair of adjacent vertices which have been assigned the same colour, we remove the coloured vertices and solve the remaining 2-colouring problem in polynomial time. Thus the overall time used is $O(f(n) \operatorname{poly}(n))$. By the Chernoff bound (Theorem 9.4), $f(n) \leq 1.938^n$ (in fact, using a tighter bound one can get $f(n) \leq 1.89^n$).
There is a nice review by Uwe Schöning, called "Algorithmics in Exponential Time", of these and other algorithms for NP-complete problems.

Lecture 6

Adding Communication Into The Mix

So far we have looked at how resouces such as random number numbers and space have an impact on our notion of how difficult it is do complete some task. We now examine the resource of communication. It turns out that this resource allows us to complete tasks in some philosophical sense much easier than if we did not have communication. We can summarize this in the sentence, once used to advertise BT telephone services, in that "It is good to talk".

The main result of this part is what we have been trying to get to as the "highlight" of the course. The fact that $\mathsf{PSPACE} = \mathsf{IP}$. The class IP of interactive proofs is akin to a lecture, we have an all seeing expert (the lecturer, or Merlin) who wants to impart some knowledge to the novice (the students, or Arthur). To do this they engage in an interactive protocol[1], so that the expertise is transferred from the lecturer to the student.

The result that $\mathsf{IP} = \mathsf{PSPACE}$ was announced on Boxing Day in 1989 by Shamir, swiftly following important prior work by Lund, Fortnow, Karloff, Nisan and others; Babai has written a nice discussion of the history of this result[2]. The exposition here of the proof is essentially based on lecture notes by Jonathan Katz[3] and Arora-Barak chapter 8; there are also some good lecture notes by Oded Goldreich[4]. The result that $\mathsf{MIP} = \mathsf{NEXP}$ is due to Babai, Fortnow and Lund and came a matter of weeks after the proof that $\mathsf{IP} = \mathsf{PSPACE}$.

An important application of these concepts is in cryptography, where we want so called zero-knowledge protocols. These are interactive protocols for which Merlin just convinces Arthur he knows the magic formulaes without revealing any details about the formulae themselves. This has many applications, most noteably in authentication devices such as chip-and-pin systems.

[1] Note this implies good students should interact in lectures

[2] `http://www.cs.princeton.edu/courses/archive/spr09/cos522/BabaiEmail.pdf`

[3] `http://www.cs.umd.edu/~jkatz/complexity/f11/lecture19.pdf`

[4] `http://eccc.hpi-web.de/resources/pdf/cc2.pdf`

6.1 Communication complexity

The theory of communication complexity studies the amount of communication required for two (or more) spatially separated parties to accomplish some task. While a simple model, it turns out to have numerous applications in theoretical computer science. As Arora and Barak note, "it strikes the elusive balance of being simple enough so that we can actually prove strong lower bounds, but general enough so that we can obtain important applications of these lower bounds".

In the simplest variant of the model, we imagine we have two players (conventionally known as Alice and Bob). Each has an input picked from some set (Alice has $x \in X$, Bob has $y \in Y$). They wish to compute some known function $f(x, y)$, where $f : X \times Y \to Z$. Unfortunately, Alice does not know Bob's input y, and Bob does not know Alice's input x, so they have to communicate in order to compute $f(x, y)$. Their goal is for Bob to output $f(x, y)$, perhaps with some probability of failure, using the smallest possible amount of communication. They have previously agreed on a protocol to compute $f(x, y)$, which proceeds as follows. Alice sends a message to Bob, based on her own input x. Based on Alice's message and his own input y, Bob chooses a message to send to Alice. Alice then replies to Bob, and this protocol continues until one of the parties (we assume Bob) outputs $f(x, y)$. See Figure 6.1 for an illustration. Note that, we assume that computation is free: Alice and Bob are both computationally unbounded. This is a reasonable model for distributed computational tasks where communication is much more expensive than local computation.

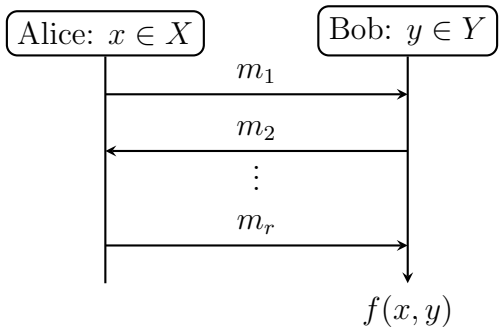


Figure 6.1: A r-message communication protocol for computing $f(x, y)$.

For a function $f : X \times Y \to Z$, we define the deterministic communication complexity of f, $D_{cc}(f)$, as the minimum amount of total communication, measured in bits, required to compute $f(x, y)$ with certainty for any possible pair of inputs x, y. We make the following initial simple observations.

- $D_{cc}(f) \leq \lceil \log_2 |X| \rceil$: to compute any function f, Alice can just send her input to Bob.
- If we require each message between the parties to be one bit long, it only changes the communication complexity of f by a constant factor. We henceforth assume this restriction.

- A protocol $\mathcal{P}$ which communicates at most k bits can thus be viewed as a binary tree with at most 2^k leaves.
- If we change the model so both Alice and Bob need to know the answer $f(x, y)$, it only changes $D_{cc}(f)$ by an additive $O(\log |Z|)$ term.

It is hopefully clear that some functions can be computed using much less communication than the trivial protocol where Alice sends all her input to Bob (or vice versa). For example, consider the function XOR2 : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ defined by $\text{XOR2}(x, y) = \bigoplus_{i=1}^n (x_i \oplus y_i)$. In other words, XOR2 is defined by taking the XOR of Alice's input with Bob's input, and then the XOR of the resulting n-bit string. Using linearity of $\oplus$, we have

$$\bigoplus_{i=1}^{n} (x_i \oplus y_i) = \left(\bigoplus_{i=1}^{n} x_i \right) \oplus \left(\bigoplus_{j=1}^{n} y_j \right).$$

Thus Alice and Bob can compute XOR2 using only one bit of communication: Alice just computes the bit $a = \bigoplus_{i=1}^n x_i$ and sends it to Bob, who has previously computed $b = \bigoplus_{j=1}^n y_j$. Bob then outputs $a \oplus b$.

The model of communication complexity was first introduced by Yao in 1979, and has developed into a rich and extensive field of study. Arora-Barak (chapter 13) has a brief discussion of communication complexity, but the bible of the subject is the book "Communication Complexity" by Kushilevitz and Nisan, from which the discussion of time-space tradeoffs in the Advanced Technical Notes is taken, and which contains many other applications of these ideas.

6.2 Interactive proofs

We now link up the two concepts of communication protocols and computational complexity by studying interpretations of complexity classes in terms of interactive protocols. An alternative way to view the complexity class NP is in terms of an interaction between two players in a game: a computationally unbounded *prover* (conventionally called Merlin) and a *verifier* (called Arthur) with access to polynomial-time computation.

We imagine that Arthur is given an input $x \in \{0,1\}^n$, and would like to determine whether $x \in \mathcal{L}$, for some language $\mathcal{L}$ not necessarily in P, but is only able to perform deterministic polynomial-time computation. However, he also has access to a wizard (Merlin), who is computationally unbounded and can hence easily determine whether $x \in \mathcal{L}$. The interaction between Merlin and Arthur consists only of a single message from Merlin to Arthur. Furthermore, and unlike the previously discussed setting of communication complexity, Merlin cannot be trusted. Thus, in order for Arthur to be convinced that $x \in \mathcal{L}$, Merlin must send him a proof of this, which Arthur can check for himself. It is easy to see that NP is precisely the class of languages which Arthur can decide using a protocol of this form.

We now generalise this idea by allowing protocols with multiple rounds, and for Arthur to have some probability of failure. An interactive proof system with k messages is defined as follows. The input string $x \in \{0,1\}^n$ is known to both the prover and the verifier before the protocol starts. The verifier also has access to a string r of random bits which the prover cannot see. During the protocol, the verifier V and prover P alternately generate messages $m_1, \ldots, m_k$ to their counterpart based on any of the previous messages in the protocol, and (in the case of the verifier) the random string r. Thus the messages produced are of the form

$$m_1 = V(x, r),\ m_2 = P(x, m_1),\ m_3 = V(x, r, m_1, m_2),\ m_4 = P(x, m_1, m_2, m_3), \ldots$$

We assume that the last message is always from the prover to the verifier, following which the verifier either accepts or rejects x. If k is even, a k-message protocol is also called a $k/2$-round protocol, where a round is a pair of messages (one from the verifier to the prover, and one in the opposite direction). We further assume that k, r and the size of each message are all polynomial in n. Figure 6.2 illustrates such a protocol.

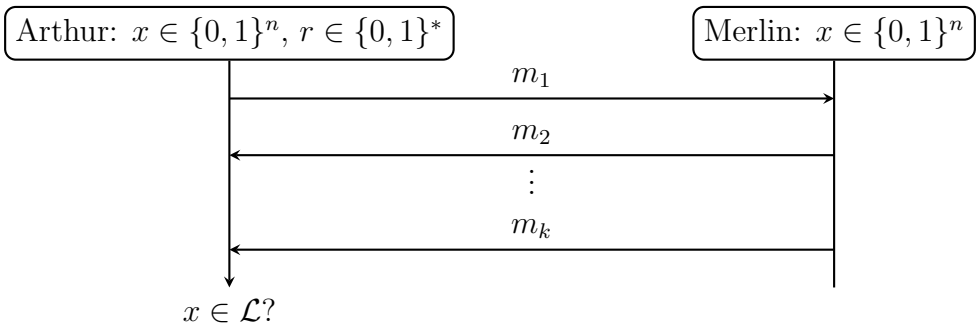


Figure 6.2: A k-message interactive proof for deciding $\mathcal{L}$.

We are now ready to define the complexity class IP.

Definition 16. *For an integer $k \geq 1$, we say that a language $\mathcal{L}$ is in $\mathsf{IP}[k]$ if there is an interactive proof system as above, with k messages between a probabilistic polynomial-time Turing machine V and a computationally unbounded prover P, such that*

- **(Completeness)** *If $x \in \mathcal{L}$ then there exists a prover P such that the probability that V accepts x is at least $2/3$,*
- **(Soundness)** *If $x \notin \mathcal{L}$ then, for all provers P, the probability that V accepts x is at most $1/3$,*

where probabilities are taken over the choice of r. We also write $\mathsf{IP} = \bigcup_{k \geq 0} \mathsf{IP}[k]$.

Observe that it is immediate from the definition that $\mathsf{BPP} = \mathsf{IP}[0]$ and $\mathsf{NP} \subseteq \mathsf{IP}[1]$. Just as with the class BPP, we first show that this definition essentially does not depend on the choice of constants $2/3$, $1/3$, as these can be made exponentially close to 1 and 0.

Theorem 47. *The class* IP *is unchanged if we replace the completeness parameter* $2/3$ *by* $1 - 2^{-n^s}$*, and the soundness parameter* $1/3$ *by* 2^{-n^s}*, for any constant* $s > 0$*.*

Proof. The verifier simply repeats the protocol m times, and accepts if at least $m/2$ of the runs would accept. Consider the prover which just independently repeats the strategy which each of the m sub-verifiers accepts with probability at least $2/3$. If $x \in \mathcal{L}$, then the verifier will accept x with probability at least $1 - 2^{-\Omega(m)}$ (by a Chernoff bound, Corollary 40).

On the other hand, if $x \notin \mathcal{L}$, we need to show that every prover strategy will fail with high probability to convince the verifier that $x \in \mathcal{L}$. Note that this includes more complicated strategies than this independent strategy previously discussed; in particular, the prover's strategy at each stage of the protocol may depend on all the previous stages. However, at each stage the probability of the verifier accepting is still upper bounded by $1/3$, because the definition of soundness above holds for *all* provers. By a similar Chernoff bound argument, this implies that the verifier will accept x with probability at most $2^{-\Omega(m)}$. □

6.3 Interactive Proofs and Polynomial Space

The following remarkable result completely characterises the interactive proof model, in terms of a rather powerful and different-looking complexity class.

Theorem 48. $\mathsf{IP} = \mathsf{PSPACE}$*.*

Theorem 48 can be split into two parts.

Theorem 49. $\mathsf{IP} \subseteq \mathsf{PSPACE}$*.*

Proof. Let $\mathcal{L} \in \mathsf{IP}$, and let be a *fixed* V be the verifier for the corresponding IP protocol for deciding $\mathcal{L}$. Assume the protocol has k messages. We would like to show that we can determine whether $x \in \mathcal{L}$, for any x, using polynomial space. It suffices to determine whether there exists a prover P such that the probability that V outputs 1 using prover P on input x is at least $2/3$; if there is such a prover, then $x \in \mathcal{L}$, otherwise $x \notin \mathcal{L}$.

Let $P(x, m_1, \ldots, m_i)$ be the maximal probability that x is accepted over all provers, given that the first i messages were $m_1, \ldots, m_i$. That is, the maximum is taken over all possible choices of prover from message $i + 1$ onwards. $P(x)$ is precisely the maximal probability that V accepts x, over all provers, while $P(x, m_1, \ldots, m_k)$ can be evaluated given only knowledge of the verifier's behaviour. For each i, we therefore compute $P(x, m_1, \ldots, m_{i-1})$ from $P(x, m_1, \ldots, m_i)$. If m_i is a message from the prover to the verifier, we have

$$P(x, m_1, \ldots, m_{i-1}) = \max_{m_i} P(x, m_1, \ldots, m_i),$$

whereas if m_i is a message from the verifier to the prover we have

$$P(x, m_1, \ldots, m_{i-1}) = \mathbb{E}_{r_i} P(x, m_1, \ldots, m_i),$$

where r_i is the string of random bits used by the verifier to determine message m_i. $P(x)$ can therefore be computed recursively using polynomial space, by a similar argument to the proof that TQBF can be solved in polynomial space.

Note we compute this probability from the fixed verifier and the input string. The resulting TM is in PSPACE (just proved) and if the probability is close to one one we conclude $x \in \mathcal{L}$, otherwise we conclude $x \notin \mathcal{L}$. □

In order to show the reverse inclusion $\mathsf{PSPACE} \subseteq \mathsf{IP}$, it suffices to give an interactive protocol for the TQBF problem; the proof of this is somewhat technical, so we prove a simpler result. Leaving the proof for TQBF to the technical notes. Define the language 3-SAT$_K$ as follows:

$$\text{3-SAT}_K = \{\phi : \phi \text{ is a 3-CNF formula with exactly } K \text{ satisfying assignments}\}.$$

This is a generalisation of the $\mathsf{co\text{-}NP}$-complete language $\overline{\text{3-SAT}}$, which is just 3-SAT$_0$, and a restriction of the #P-complete problem #3-SAT. We will show that 3-SAT$_K$ is in IP for any K; in particular, this implies $\mathsf{co\text{-}NP} \subseteq \mathsf{IP}$.

Arithmetizing 3-SAT$_K$: Before proving 3-SAT$_K$ is in IP we need to recap on *arithmetisation*, which we discussed discussed briefly in the first lecture. Recall, arithmetisation is a process establishing a link between boolean formulas and polynomials over some finite field $\mathbb{F}$. Given a 3-CNF formula $\phi(x_1, \ldots, x_n)$ with m clauses, introduce variables $X_1, \ldots, X_n \in \mathbb{F}$. For each clause, write a polynomial of degree at most 3 by mapping $x_i \mapsto (1 - X_i)$, $\neg x_i \mapsto X_i$, taking the product and subtracting the result from 1. For example,

$$x_1 \vee \neg x_4 \vee x_5 \mapsto 1 - (1 - X_1)X_4(1 - X_5),$$

and let the polynomial for the j'th clause be denoted $p_j(X_1, \ldots, X_n)$. For each possible assignment of 0's and 1's to $X_1, \ldots, X_n$, $p_j(X_1, \ldots, X_n) = 1$ if the j'th clause is satisfied by that assignment, and $p_j(X_1, \ldots, X_n) = 0$ otherwise. Multiplying these polynomials together, we get

$$P_\phi(X_1, \ldots, X_n) = \prod_{j=1}^{m} p_j(X_1, \ldots, X_n),$$

which evaluates to 1 on satisfying assignments $X_1, \ldots, X_n$ and 0 on unsatisfying assignments. Thus P_ϕ is a polynomial of degree at most $3m$.

We now use arithmetisation to prove the following claim.

Theorem 50. *3-SAT$_K$ $\in \mathsf{IP}$.*

Given input ϕ, we begin by constructing the polynomial P_ϕ. The number of satisfying assignments of ϕ is precisely

$$N_\phi := \sum_{X_1 \in \{0,1\}} \sum_{X_2 \in \{0,1\}} \cdots \sum_{X_n \in \{0,1\}} P_\phi(X_1, \ldots, X_n).$$

The prover wishes to convince the verifier that $N_\phi = K$. The prover first sends the verifier a prime p such that $2^n < p \leq 2^{2n}$; the verifier can check that p is prime using a polynomial-time primality test (e.g. the Miller-Rabin test discussed previously, or a deterministic test for primality). From now on, all computations are done over the field $\mathbb{F}_p$. As $0 \leq N_\phi < p$, $N_\phi = 0$ if and only if $N_\phi \equiv 0$ modulo p.

We now give a general protocol for testing the following claim: given a prime p, an element $K \in \mathbb{F}_p$, and a degree d polynomial $g(X_1, \ldots, X_n) : \mathbb{F}_p^n \to \mathbb{F}_p$,

$$\sum_{X_1 \in \{0,1\}} \sum_{X_2 \in \{0,1\}} \cdots \sum_{X_n \in \{0,1\}} g(X_1, \ldots, X_n) = K. \tag{6.1}$$

We assume that the verifier can evaluate $g(X_1, \ldots, X_n)$ for any $X_1, \ldots, X_n$ in time poly(n) (this is certainly true for $g = P_\phi$). For each assignment of values $b_2, \ldots, b_n \in \{0, 1\}$ to $X_2, \ldots, X_n$, $g(X_1, b_2, \ldots, b_n)$ is a polynomial of degree at most d in the single variable X_1. Thus

$$h(X_1) := \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} g(X_1, b_2, \ldots, b_n)$$

is also a degree d polynomial, and if claim (6.1) holds, then $h(0) + h(1) = K$. In order to check this claim, we perform the following recursive protocol.

- If $n = 1$, the verifier checks that $g(0) + g(1) = K$. If so, he accepts; otherwise, he rejects. If $n \geq 2$, the verifier asks the prover to send a description of the polynomial $h(X_1)$.
- The prover sends a polynomial $\tilde{h}(X_1)$ (if the prover is honest, $\tilde{h}(X_1) = h(X_1)$).
- The verifier rejects if $\tilde{h}(0) + \tilde{h}(1) \neq K$. Otherwise, he picks $r \in \mathbb{F}_p$ uniformly at random, and uses the same protocol recursively to check that

$$\sum_{X_2 \in \{0,1\}} \cdots \sum_{X_n \in \{0,1\}} g(r, X_2, \ldots, X_n) = \tilde{h}(r).$$

Lemma 51. *If (6.1) is false, then the verifier rejects with probability at least* $(1 - d/p)^n$.

Proof. Assuming that (6.1) is false, we prove the lemma by induction on n. For the base case $n = 1$, if $g(0) + g(1) \neq K$ the verifier rejects with certainty. In the first round, the prover is supposed to return $h(X_1)$. If he does so, then as $h(0) + h(1) \neq K$, the verifier rejects with certainty. So assume $\tilde{h}(X_1) \neq h(X_1)$. Then $\tilde{h}(X_1) - h(X_1)$ is a nonzero polynomial of degree at most d, and hence has at most d roots. Thus there are at most d values r such that $\tilde{h}(r) = h(r)$. Hence, when the verifier picks a random r, $\Pr[\tilde{h}(r) \neq h(r)] \geq 1 - d/p$. If $\tilde{h}(r) \neq h(r)$, then at the next recursive step the prover has an incorrect claim to prove. By the inductive hypothesis, he fails to convince the verifier with probability at least $(1 - d/p)^{n-1}$. Hence the probability that the verifier rejects is at least $(1-d/p)(1-d/p)^{n-1} = (1-d/p)^n$. □

This lemma implies Theorem 50, because we have $(1 - d/p)^n \geq 1 - nd/p \geq 1 - 3mn/2^n$, which is very close to 1 and in particular larger than 2/3.

6.4 Zero-Knowledge

We end this set of main notes by looking at a variant of IP called CZK, for Zero-Knowledge interactive protocols. Here the idea is for the verifier to be convinced of the truth of the statement being proved, and nothing else. In particular it does not learn the witness.

You use CZK protocols every day; on your phone, in a back etc. The "witness", say, is the secret inside your bank card which proves that it is a valid bank card and that you have unlocked it with your PIN. The verifier is the bank ATM machine, it engages in a protocol with the card to learn the bit of the information that the card has the witness. But it does not learn the witness, or in fact anything about the witness.

So whilst some areas of Computer Science are about knowing something, in zero-knowledge we have to define *not* knowing something. The basic idea is the so-called *simulation* paradigm.

Think of a valid prover-verifier conversation $(q_1, r_1, q_2, r_2, \ldots, r_n)$ where q_i are the questions of the verifier and r_i are the responses of the prover. We call this sequence a "transcript". If this is the output of a protocol in IP then the prover knows the statement is correct (e.g. he may have a witness w), and the verifier learns one bit of information. Both parties have access to $x \in \mathcal{L}$. The next task is to work out whether the prover learns anything else. In other words is the transcript useful to him to do anything else with it.

Now suppose there is an algorithm S called a simulator, which on input of x outputs a "fake" transcript $(q'_1, r'_1, q'_2, r'_2, \ldots, r'_n)$. We say "fake" as it is not the output of the protocol, and the simulator is not given the witness, it is only given x. Of course it the witness was easy to compute then he could do this, but then so could the verifier!

Now suppose that the fake transcripts and the real transcripts "look the same" to the verifier, i.e. he cannot tell them apart. So if we has an algorithm to learn something from the transcripts he could run this algorithm on the "fake" ones. In which case he does not need the real ones. Think about this for a minute and you see that this means he does in fact learn nothing bar the fact that the prover knows the statement is correct. i.e. the protocol is zero-knowledge.

So if we have a protocol for a language $\mathcal{L} \in$ IP for which there is a simulator S such that the real transcripts of the protocol look identical to the output of S then we say the protocol is zero-knowledge. The class CZK is the set of all languages which are recognized by a protocol which is zero-knowledge. We know that CZK is quite big, in fact NP $\subseteq$ CZK $\subseteq$ IP. Indeed, if one way functions exist, i.e. crypto is at all possible, then CZK = IP.

6.5 Supplementary Notes

6.5.1 Graph Non-Isomorphism and IP

Although IP is at least as powerful as NP, it is not clear that IP should be any more powerful than this (e.g. compare the conjecture that BPP = P). To build intuition, we give an example of an interactive proof for a problem which is not known to be in NP: Graph Non-Isomorphism. This problem is the complement of the Graph Isomorphism problem mentioned in Section 2.4. Given the description of two undirected graphs G and H, we have to accept if there is no permutation π

of the vertices of G such that $\pi(G) = H$ (see Figure 6.3). The GRAPH ISOMORPHISM problem is in NP (and thus GRAPH NON-ISOMORPHISM is in co-NP), as a certificate of isomorphism between G and H is just the permutation π. However, it is not obvious how to certify that no such π exists, and it is not known whether GRAPH NON-ISOMORPHISM is in NP.

Figure 6.3: Two isomorphic graphs.

Nevertheless, we can give an interactive proof for GRAPH NON-ISOMORPHISM. The protocol proceeds as follows.

1. Arthur picks one of G and H at random, applies a random permutation π to the vertices and sends the permuted graph P to Merlin.

2. Merlin attempts to identify which of the graphs G and H was used to produce P and sends this information back to Arthur.

3. Arthur accepts if Merlin's reply is correct, and rejects otherwise.

We now prove that this protocol works. If G and H are not isomorphic, then given P, Merlin can identify whether $P = \pi(G)$ or $P = \pi(H)$ (he has unbounded computational power and in particular can solve GRAPH ISOMORPHISM!). Thus Merlin can reply with the correct answer and Arthur accepts with certainty. On the other hand, if G and H are isomorphic, then $\pi(G)$ is indistinguishable from $\pi(H)$, so Merlin cannot guess which graph was used to produce P with probability greater than $1/2$ (which he can achieve by just picking G or H at random). Thus Arthur incorrectly accepts isomorphic graphs with probability at most $1/2$. This can be reduced to $1/4$ (to fit into the definition of IP) by performing the protocol twice and accepting only if both repetitions accept.

Self Test Question:
Let p be prime. An integer y is said to be a quadratic residue modulo p if there exists x such that $y \equiv x^2 \pmod{p}$. Prove (directly) that the following language is in IP.

$$\text{QNR} = \{(y, p) : y \text{ is not a quadratic residue modulo } p\}.$$

Solution:
Consider the following protocol. Arthur picks a random integer $r \in \{0, \ldots, p-1\}$ and a random bit $c \in \{0, 1\}$. If $c = 0$, Arthur sends Merlin $r^2 \pmod{p}$, and if $c = 1$, Arthur sends Merlin $yr^2 \pmod{p}$. Arthur asks Merlin to return the value of c. We claim that, if y is not a quadratic residue, Merlin can return the right answer with certainty; however, if y is a quadratic residue, Merlin cannot succeed with probability greater than $1/2$.
To see this, observe that the set of quadratic residues modulo p forms a group. Thus, if y is a quadratic residue, the values of r^2 and yr^2 are equally distributed, so Merlin cannot guess which is which with probability greater than $1/2$. But if y is not a quadratic residue, neither is yr^2, so Merlin can distinguish these cases with certainty.

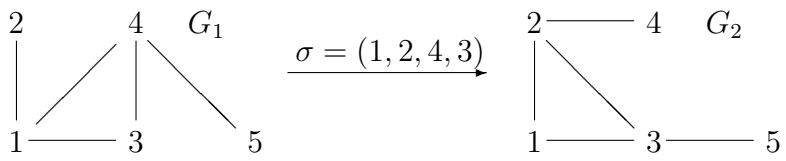
6.5.2 Zero Knowledge

The classic example of a zero-knowledge proof is based on the graph isomorphism problem. Given two graphs G_1 and G_2, with the same number of vertices, we say that the two graphs are isomorphic if there is a relabelling (i.e. a permutation) of the vertices of one graph which produces the second graph. This relabelling ϕ is called the graph isomorphism which is denoted by

$$\phi : G_1 \longrightarrow G_2.$$

It is a hard computational problem to determine a graph isomorphism between two graphs. As a running example consider the two graphs in Figure 6.4, linked by the permutation $\phi = (1, 2, 4, 3)$.

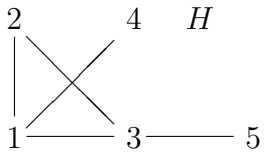
Figure 6.4: Example graph isomorphism



Suppose Peggy knows the graph isomorphism ϕ between two public graphs G_1 and G_2, so we have $G_2 = \phi(G_1)$. We call ϕ the prover's private input, whilst the graphs G_1 and G_2 are the public or common input. Peggy wishes to convince Victor that she knows the graph isomorphism, without revealing to Victor the precise nature of the graph isomorphism. This is done using the following zero-knowledge proof.

- Peggy takes the graph G_2 and applies a secret random permutation ψ to the vertices of G_2 to produce another isomorphic graph $H \leftarrow \psi(G_2)$. In our running example we take $\psi = (1, 2)$, the isomorphic graph H is then given by Figure 6.5.
- Peggy now publishes H as a **commitment**, she of course knows the following secret graph

Figure 6.5: Peggy's committed graph



isomorphisms

$$\phi : G_1 \longrightarrow G_2,$$
$$\psi : G_2 \longrightarrow H,$$
$$\psi \circ \phi : G_1 \longrightarrow H.$$

- Victor now gives Peggy a **challenge**, he selects $b \in \{1, 2\}$ and asks for the graph isomorphism between G_b and H.
- Peggy now gives her **response** by returning either $\chi = \psi$ or $\chi = \psi \circ \phi$, depending on the value of b.
- Victor now verifies whether $\chi(G_b) = H$

The transcript of the protocol then looks like

$$P \longrightarrow V : H,$$
$$V \longrightarrow P : b,$$
$$P \longrightarrow V : \chi.$$

In our example if Victor chooses $b = 2$ then Peggy simply needs to publish ψ. However, if Victor chooses $b = 1$ then Peggy publishes

$$\psi \circ \phi = (1, 2) \circ (1, 2, 4, 3) = (2, 4, 3).$$

We can then see that $(2, 4, 3)$ is the permutation which maps graph G_1 onto graph H. But to compute this we needed to know the hidden isomorphism ϕ. Thus when $b = 2$ then Victor is checking whether Peggy is honest in her commitment, whilst if $b = 1$ we are checking whether Peggy is honest in her claim to know the isomorphism from G_1 to G_2.

If Peggy does not know the graph isomorphism ϕ then she will need to know, before Victor gives his challenge, the graph G_b which Victor is going to pick. Hence, if Peggy is cheating she will only be able to respond to Victor correctly 50 percent of the time. So, repeating the above protocol a number of times, a non-cheating Peggy will be able to convince Victor that she really does know the graph isomorphism, with a small probability of error of Victor being convinced incorrectly.

Now we need to determine whether Victor learns anything from running the protocol, i.e. is Peggy's proof really zero-knowledge? We first notice that Peggy needs to produce a different value

of H on every run of the protocol, otherwise Victor can trivially cheat. We assume therefore that this does not happen.

One way to see whether Victor has learnt something after running the protocol is to look at the transcript of the protocol and ask after having seen the transcript whether Victor has gained any knowledge, or for that matter whether anyone looking at the protocol but not interacting learns anything. One way to see that Victor has not learnt anything is to see that Victor could have written down a valid protocol transcript without interacting with Peggy at all. Hence, Victor cannot use the protocol transcript to convince someone else that he knows Peggy's secret isomorphism. He cannot even use the protocol transcript to convince another party that Peggy knows the secret graph isomorphism.

Victor can produce a valid protocol transcript using the following *simulation*:

- $b \leftarrow \{1, 2\}$.
- Generate a random isomorphism χ of the graph G_b to produce the graph H.
- Output the transcript

$$\begin{aligned} P &\longrightarrow V : H, \\ V &\longrightarrow P : b, \\ P &\longrightarrow V : \chi. \end{aligned}$$

Hence, the interactive nature of the protocol means that it is a zero-knowledge proof.

Clearly two basic properties of an interactive proof system are

- **Completeness:** If Peggy really knows the thing being proved, then Victor should accept her proof with probability one.
- **Soundness:** If Peggy does not know the thing being proved, then Victor should only have a small probability of actually accepting the proof.

Just as with commitment schemes we can divide zero-knowledge protocols into categories depending on whether they are secure with respect to computationally bounded or unbounded adversaries. We usually assume that Victor is a polynomially bounded party, whilst Peggy is unbounded. In the above protocol based on graph isomorphism we saw that the soundness probability was equal to one half. Hence, we needed to repeat the protocol a number of times to reduce this to something small.

The zero-knowledge property we have already noted is related to the concept of a simulation. Suppose the set of valid transcripts (produced by true protocol runs) is denoted by $\mathcal{V}$ and let the set of possible simulations be denoted by $\mathcal{S}$. The security is therefore related to how much like the set $\mathcal{V}$ is the set $\mathcal{S}$.

A zero-knowledge proof is said to have perfect zero-knowledge if the two sets $\mathcal{V}$ and $\mathcal{S}$ are essentially identical, in which case we write $\mathcal{V} = \mathcal{S}$. If the two sets have "small" statistical distance, but can not otherwise be distinguished by an all powerful adversary we say we have statistical zero knowledge, and we write $\mathcal{V} \approx_s \mathcal{S}$. If the two sets are only indistinguishable by a computationally bounded adversary we say that the zero-knowledge proof has computational zero-knowledge, and we write $\mathcal{V} \approx_c \mathcal{S}$.

So the question arises as to what can be shown in zero-knowledge. Above we showed that the knowledge of whether two graphs are isomorphic can be shown in zero-knowledge. Thus the decision problem of `Graph Isomorphism` lies in the set of all decision problems which can be proven in zero-knowledge. But `Graph Ismorphism` is believed to lie between the complexity classes P and NP-complete, i.e. it can neither be solved in polynomial time, yet neither is it NP-complete.

We can think of NP problems as those problems for which there is a witness (or proof) which can be produced by an all powerful prover, but which a polynomially bounded verifier can verify the proof. However, for the class of NP problems the prover and the verifier do not interact, i.e. the proof is produced and then the verifier verifies it.

If we allow interation then something quite amazing happens. Consider an all powerful prover who interacts with a polynomially bounded verifier. We wish the prover to convince the verifier of the validity of some statement. This is exactly as we had in the previous section except that we only require the completeness and soundness properties, i.e. we do not require the zero-knowledge property. The decision problems which can be proved to be true in such a manner form the complexity class of *interactive proofs*, or IP It can be shown that the complexity class IP is equal to the complexity class PSPACE, i.e. the set of all decision problems which can be solved using polynomial space. It is widely believed that NP$\subsetneq$PSPACE, which implies that having interaction really gives us something extra.

So what happens to interactive proofs when we add in the zero-knowledge requirement? We can define a complexity class CZK of all decision problems which can be verified to be true using a computational zero-knowledge proof. We have already shown that the problem of `Graph Isomorhism` lies in CZK, but this might not include all of the NP problems. However, since 3-colourability is NP-complete, we have the following elegant proof that NP $\subset$ CZK,

Theorem 52. *The problem of* 3*-colourability of a graph lies in* CZK*, assuming a computationally hiding commitment scheme exists.*

Proof. Consider a graph $G = (V, E)$ in which the prover knows a colouring ψ of the G, i.e. a map $\psi : V \to \{1, 2, 3\}$ such that $\psi(v_1) \neq \psi(v_2)$ if $(v_1, v_2) \in E$. The prover first selects a commitment scheme $C(x; r)$ and a random permutation π of the set $\{1, 2, 3\}$. Note, the function $\pi(\psi(v))$ defines another three colouring of the graph. Now the prover commits to this second three colouring by sending to the verifier the commitments

$$c_i = C\left(\pi(\psi(v_i)); r_i\right) \text{ for all } v_i \in V.$$

The verifier then selects a random edge $(v_i, v_j) \in E$ and sends this to the prover. The prover now decommits to the values of

$$\pi(\psi(v_i)) \text{ and } \pi(\psi(v_j)),$$

and the verifier checks that

$$\pi(\psi(v_i)) \neq \pi(\psi(v_j)).$$

We now turn to the three required properties of a zero-knowledge proof.

Completeness: The above protocol is complete since any valid prover will get the verifier to accept with probability one.
Soundness: If we have a cheating prover then at least one edge is invalid, and with probability at least $1/|E|$ the verifier will select an invalid edge. Thus with probability at most $1 - 1/|E|$ a

cheating prover will get a verifier to accept. By repeating the above proof many times one can reduce this probability to as low a value as we require.
Zero-Knowledge: Assuming the commitment scheme is computationally hiding the obvious simulation and the real protocol will be computationally indistinguishable. □

Notice, that this is a very powerful result. It says that virtually any statement which is like to come up in cryptography can be proved in zero-knowledge. Clearly the above proof would not provide a practical implementation, but at least we know that very powerful tools can be applied. In the next section we turn to more practical proofs which can be applied in practice. But before doing that we note that the above result can be extended even further

Theorem 53. *Assuming one-way functions exist then* $\mathsf{CZK} = \mathsf{IP}$*, and hence* $\mathsf{CZK} = \mathsf{PSPACE}$*.*

6.6 Advanced Technical Notes

6.6.1 Modifications to the interactive proof model

We have seen that allowing interaction and a very small probability of failure gives us the ability to solve any problem in the class PSPACE. This result is perhaps additionally surprising because, if we change completeness and soundness by only an exponentially small amount, we obtain precisely the class NP. Indeed, if we let dIP be the class obtained from Definition 16 by replacing the completeness parameter 2/3 by 1, and the soundness parameter 1/3 by 0, we have the following result.

Theorem 54. $\mathsf{dIP} = \mathsf{NP}$*.*

Proof. The inclusion $\mathsf{NP} \subseteq \mathsf{dIP}$ is immediate: as discussed at the start of this section, NP is precisely dIP restricted to a single message. For the other direction, we prove that if $\mathcal{L} \in \mathsf{dIP}$, in fact $\mathcal{L} \in \mathsf{NP}$. If V is the dIP verifier for $\mathcal{L}$, a certificate that $x \in \mathcal{L}$ is a sequence of messages $m_1, \ldots, m_k$ between the prover and V that causes V to accept. To verify this certificate, simply check that $V(x) = m_1$, $V(x, m_1, m_2) = m_3$, etc., and that $V(x, m_1, \ldots, m_k) = 1$. If $x \in \mathcal{L}$, there exists such a sequence of messages. But if such a sequence exists, then $x \in \mathcal{L}$, by defining a prover P such that $P(x, m_1) = m_2$, $P(x, m_1, m_2, m_3) = m_4$, etc. □

One could also consider an extension of the model of interactive proofs to multiple provers. This is conceivably more powerful than the class IP, because the verifier can perform consistency checks across different provers. That is, if one prover is trying to trick Arthur, he might be able to determine this by using another prover to double-check the first's answers. Indeed, if we define the class MIP in exactly the same way as IP, but allow the verifier to communicate with *two* provers rather than one, we have the following result, which is arguably even more remarkable than $\mathsf{IP} = \mathsf{PSPACE}$ and we state without proof.

Theorem 55. $\mathsf{MIP} = \mathsf{NEXP}$*.*

It can also be shown that extending the model further, to more than two provers, does not lead to any additional power: MIP with k provers is equal to MIP with 2 provers, for any $k = \text{poly}(n)$.

6.6.2 The rank lower bound

We would like to analyse communication protocols mathematically. Define the *communication matrix* of $f : X \times Y \to Z$ as the $|X| \times |Y|$ matrix M^f where $M^f_{xy} = f(x, y)$. The rows of M^f are indexed by Alice's inputs $x \in X$, and the columns by Bob's inputs $y \in Y$. We will analyse an arbitrary communication protocol $\mathcal{P}$ for f in terms of M^f.

A (combinatorial) rectangle in M^f is a submatrix of M^f corresponding to entries in $A \times B$, where $A \subseteq X$, $B \subseteq Y$. Note that the rows and columns of M^f are not required to be consecutive, i.e. there may be gaps. We say that $A \times B$ is monochromatic if, for all $x \in A$, $y \in B$, M^f_{xy} is constant. We now observe that any protocol $\mathcal{P}$ partitions M^f into combinatorial rectangles, which follows by induction. Assume that after some number of steps of the protocol, Alice and Bob know that the input is contained in some rectangle R (this is clearly true at the start because $X \times Y$ is a rectangle). At the next step in the protocol, one player sends a bit to the other. The choice of this bit, which depends on that player's own input, partitions R into two rectangles R_0, R_1. When the other player receives the bit, both players know that their joint inputs are contained in either R_0 or R_1.

$$\left(\begin{array}{cc|cc} 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{array}\right)$$

Figure 6.6: A completed protocol partitions a communication matrix into monochromatic rectangles.

At every stage of the protocol $\mathcal{P}$, each rectangle in the corresponding partition gives a subset of inputs for which the communication pattern between Alice and Bob has so far been identical. Thus, if $\mathcal{P}$ is to compute $f(x, y)$ successfully for all x and y, it must be the case that each rectangle in the final partition corresponding to $\mathcal{P}$ is monochromatic. Hence, if $\mathcal{P}$ computes f and communicates k bits in total, this gives a partition of M^f into at most 2^k monochromatic rectangles, as shown in Figure 6.6. We will use this simple idea to prove the following lower bound on communication complexity in terms of matrix rank (over the reals).

Theorem 56. *For any function $f : X \times Y \to \{0, 1\}$, $D_{cc}(f) \geq \lceil \log_2 \operatorname{rank}(M^f) \rceil$.*

Proof. Let $\mathcal{P}$ be a protocol for f that communicates k bits. Let L_1 be the set of leaves of $\mathcal{P}$ (viewed as a tree) such that the output is 1. For each $\ell \in L_1$, let R_ℓ be the rectangle given by the subset of the inputs that reach the leaf ℓ, and define the matrix M^ℓ by

$$M^\ell_{xy} = \begin{cases} 1 & \text{if } (x, y) \in R_\ell \\ 0 & \text{if } (x, y) \notin R_\ell. \end{cases}$$

As the rectangles R_ℓ partition $X \times Y$, we have $M^f = \sum_{\ell \in L_1} M^\ell$. By subadditivity of the rank, this implies that

$$\operatorname{rank}(M^f) \leq \sum_{\ell \in L_1} \operatorname{rank}(M^\ell).$$

But for each ℓ, $\text{rank}(M^\ell) = 1$, so $\text{rank}(M^f) \leq |L_1| \leq 2^k$. □

It is a long-standing but unproven conjecture that Theorem 56 is close to tight.

Conjecture 57 (Log-Rank Conjecture)**.** *For any function $f : X \times Y \to \{0,1\}$, $D_{cc}(f) \leq \text{poly}(\log \text{rank}(M^f))$.*

We now give a couple of examples demonstrating the power of this bound.

Theorem 58. *For any set X, let $EQ : X \times X \to \{0,1\}$ be the equality function,*

$$EQ(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise.} \end{cases}$$

Then $D_{cc}(EQ) = \lceil \log_2 |X| \rceil$.

Proof. The communication matrix M^{EQ} is simply the $|X| \times |X|$ identity matrix, which has rank $|X|$. For the upper bound, Alice can just send all her input to Bob. □

Theorem 59. *Let $IP : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be the inner product function over $\mathbb{F}_2$: $IP(x,y) = x \cdot y = \bigoplus_{i=1}^n x_i y_i$. Then $D_{cc}(IP) = n$.*

In order to prove this result, it will be convenient to introduce an important set of functions: the characters of the group $\mathbb{Z}_2^n$. These are the 2^n functions $\chi_S : \{0,1\}^n \to \{\pm 1\}$, $S \subseteq [n]$, defined by

$$\chi_S(x) = (-1)^{\sum_{i \in S} x_i}.$$

For example, $\chi_\emptyset(x) = 1$ for all x, and $\chi_{[n]}(x) = (-1)^{\text{wt}(x)}$. These functions satisfy the following important orthogonality relations.

Lemma 60. *For any $S, T \subseteq [n]$,*

$$\sum_{x \in \{0,1\}^n} \chi_S(x)\chi_T(x) = \begin{cases} 2^n & \text{if } S = T \\ 0 & \text{otherwise.} \end{cases}$$

Proof. We have

$$\sum_{x \in \{0,1\}^n} \chi_S(x)\chi_T(x) = \sum_{x \in \{0,1\}^n} (-1)^{\sum_{i \in S} x_i + \sum_{j \in T} x_j} = \sum_{x \in \{0,1\}^n} (-1)^{\sum_{i \in S \Delta T} x_i},$$

recalling that $S \Delta T$ denotes the symmetric difference between the sets S and T, i.e. the set of integers i in either S or T, but not both. The lemma follows from the claim that, for any $S \neq \emptyset$,

$$|\{x : \bigoplus_{i \in S} x_i = 0\}| = |\{x : \bigoplus_{i \in S} x_i = 1\}|.$$

But this claim holds because there is a bijection between these two sets: for each x such that $\bigoplus_{i \in S} x_i = 0$, flipping bit j of x, for arbitrary $j \in S$, gives x' such that $\bigoplus_{i \in S} x'_i = 1$. □

Proof of Theorem 59. Consider the matrix N defined by $N_{xy} = 1 - 2M^{\text{IP}}_{xy}$. As the all 1's matrix has rank 1, $\text{rank}(M^{\text{IP}}) \geq \text{rank}(N) - 1$. We have

$$N_{xy} = (-1)^{\bigoplus_{i=1}^n x_i y_i}.$$

Thus, if we identify bit-strings $y \in \{0,1\}^n$ with subsets $Y \subseteq [n]$ in the obvious way ($y_i = 1 \Leftrightarrow i \in Y$), we see that the rows of N are given by characters of $\mathbb{Z}_2^n$. By Lemma 60, the rows of N are all orthogonal, so N has rank 2^n. Thus $D_{cc}(\text{IP}) \geq \lceil \log_2(2^n - 1) \rceil$, which equals n for $n \geq 2$. The special case $n = 1$ can be verified directly. □

6.6.3 Randomised communication complexity

We can define a randomised variant of communication complexity. The basic idea is that Alice and Bob's protocol $\mathcal{P}$ can depend on a string r of uniformly random bits: at each step of the protocol, each player decides what to reply based on r, as well as their inputs and the communication so far. However, there is an additional complication: do Alice and Bob each see the other's random coin flips (i.e. the randomness r is public), or do they each not know what the other's random coin flips are (i.e. r is split into two strings r_A, r_B)? These two models could potentially give rise to different measures of complexity. The second model is called "private" to contrast with "public", but note that there is no implication that Alice's random string should be kept secret from Bob, or vice versa.

For any function $f : X \times Y \to Z$, we define $R_{cc}(f)$ as the minimum amount of total communication required to compute $f(x, y)$ with success probability at least $2/3$ on any pair of inputs (x, y), where Alice and Bob are each given a *private* random bit-string; we define $R_{cc}^{pub}(f)$ similarly, but give Alice and Bob a public random bit-string. As with amplification of BPP algorithms, the model is not significantly changed in either setting if the success criterion is changed from $2/3$ to an arbitrary constant strictly greater than $1/2$. Observe that the length of the random string does not feature in the calculation of complexity; that is, it can be arbitrarily long. The following example illustrates the power of (public) randomness.

Theorem 61. *Let $EQ : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be the equality function on two n-bit strings. Then $R_{cc}^{pub}(EQ) \leq 2$.*

Proof. Consider the following protocol $\mathcal{P}$. Alice and Bob jointly choose a uniformly random string $r \in \{0,1\}^n$. Alice sends Bob the single bit $x \cdot r = \bigoplus_{i=1}^n x_i r_i$. Bob computes $y \cdot r$ and outputs 1 if $x \cdot r = y \cdot r$, and 0 otherwise. If $x = y$, then Bob will always output 1. On the other hand, if $x \neq y$, by Lemma 60 $\Pr_r[x \cdot r \neq y \cdot r] = 1/2$, so Bob will output 0 with probability $1/2$. If the players carry out $\mathcal{P}$ twice (using independently chosen r), and Bob outputs 1 if and only if both runs output 1, the probability of failure is at most $1/4$, which is strictly less than $1/3$. □

We thus see that $R_{cc}^{pub}(f)$ can be almost arbitrarily small compared with $D_{cc}(f)$. It is clear that for any f, $R_{cc}^{pub}(f) \leq R_{cc}(f)$: given a public random string r, the players can simply choose to divide it into substrings r_A, r_B such that Alice only looks at r_A, and Bob only looks at r_B. On the other hand, we now show that $R_{cc}^{pub}(f)$ cannot be too much smaller than $R_{cc}(f)$.

Theorem 62 (Newman's Theorem)**.** *For any function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, $R_{cc}(f) = O(R_{cc}^{pub}(f) + \log n)$.*

Proof. It suffices to show that any public coin protocol $\mathcal{P}$ can be transformed into another public coin protocol $\mathcal{P}'$ which has the same communication complexity, and such that $\mathcal{P}'$ uses only $O(\log n)$ random bits. Then, to make $\mathcal{P}'$ into a private coin protocol, Alice can just generate these random bits herself and send them to Bob before the protocol starts.

Let $Z(x, y, r)$ be the random variable which is equal to 1 if $\mathcal{P}$ is wrong on input (x, y) and random string r (i.e. $\mathcal{P}$ outputs 0 when it should output 1, or vice versa), and is equal to 0 otherwise. As $\mathcal{P}$ computes f with failure probability at most $1/3$, $\mathbb{E}_r[Z(x, y, r)] \leq 1/3$ for all (x, y). We now, iteratively, define a protocol which uses fewer random bits.

For some integer $t > 0$, let $r_1, \ldots, r_t$ be arbitrary bit-strings, and define the protocol $\mathcal{P}_{r_1,\ldots,r_t}$ as follows: Alice and Bob choose $1 \leq i \leq t$ uniformly at random, and then carry out protocol $\mathcal{P}$ with

r_i as their public random string. We claim that, for any $\delta > 0$, if we take $t = O(n/\delta^2)$ there exist strings $r_1, \ldots, r_t$ such that $\mathbb{E}_i[Z(x, y, r_i)] \leq 1/3+\delta$, for all (x, y). This claim will imply the theorem: Alice and Bob use the protocol $\mathcal{P}_{r_1,\ldots,r_t}$ to compute f, which uses $O(\log n+\log(1/\delta))$ public random bits and achieves success probability at least $2/3 - \delta$. Taking δ to be a small constant, we have the theorem.

It remains to prove that, for $t = O(n/\delta^2)$, there exist strings $r_1, \ldots, r_t$ such that $\mathbb{E}_i[Z(x, y, r_i)] \leq 1/3 + \delta$, for all (x, y). To show such a set of strings exist, we use the probabilistic method and pick the strings at random. For a particular input pair (x, y), the probability (over strings $r_1, \ldots, r_t$) that the failure probability is greater than $1/3 + \delta$ is

$$\Pr_{r_1,\ldots,r_t}[\mathbb{E}_i[Z(x, y, r_i)] > 1/3 + \delta] = \Pr_{r_1,\ldots,r_t}\left[\left(\frac{1}{t}\sum_{i=1}^{t} Z(x, y, r_i)\right) - 1/3 > \delta\right].$$

For each i, $\mathbb{E}_{r_i} Z(x, y, r_i) \leq 1/3$. Thus, applying the Chernoff bound (Corollary 40), we get

$$\Pr_{r_1,\ldots,r_t}[\mathbb{E}_i[Z(x, y, r_i)] > 1/3 + \delta] \leq e^{-\delta^2 t/3}.$$

Choosing $t = O(n/\delta^2)$, this is smaller than 2^{-2n}. Thus, taking a union bound over all pairs (x, y), the probability over $r_1, \ldots, r_t$ that there exists some (x, y) that $\mathbb{E}_i[Z(x, y, r_i)] > 1/3 + \delta$ is strictly less than 1, so there must exist *some* choices $r_1, \ldots, r_t$ such that $\mathbb{E}_i[Z(x, y, r_i)] \leq 1/3 + \delta$. This proves the claim. □

Combining Theorems 61 and 62 implies that there exists a private coin protocol for the equality function $\mathrm{EQ} : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$ which uses only $O(\log n)$ bits of communication, but does not give an explicit description of such a protocol. We now describe an explicit private-coin protocol which still uses only $O(\log n)$ bits of communication.

Let p be a prime number such that $n^2 < p < 2n^2$ (it follows from number-theoretic arguments which we omit that one can always find such a p). If Alice's input is $a \in \{0, 1\}^n$ and Bob's input is $b \in \{0, 1\}^n$, we think of these as polynomials $A, B \in \mathbb{F}_p[x]$:

$$A(x) = a_1 + a_2 x + \cdots + a_n x^{n-1},\ B(x) = b_1 + b_2 x + \cdots + b_n x^{n-1}.$$

The protocol proceeds as follows: Alice picks $r \in \mathbb{F}_p$ uniformly at random and sends Bob r and $A(r)$; Bob then outputs 1 if $A(r) = B(r)$, and 0 otherwise. The number of bits sent is thus $2\lceil \log_2 p \rceil = O(\log n)$. If $a = b$, then $A(x) = B(x)$ for all $x \in \mathbb{F}_p$, so Bob always outputs 1. If $a \neq b$ then A and B are distinct polynomials of degree $n - 1$. This means that they can be equal on at most $n - 1$ elements of $\mathbb{F}_p$ (to see this, note that $A - B$ is a degree $n - 1$ polynomial, which can have at most $n - 1$ roots). Thus the probability that Bob incorrectly outputs 1 is at most $(n - 1)/p \leq 1/n$.

6.6.4 Lower bounds on randomised communication complexity

It is also possible to prove lower bounds on randomised communication protocols. To do so, we introduce the model of distributional communication complexity, in which the communication protocol is deterministic, but the inputs are picked according to some probability distribution.

Consider $f : X \times Y \to Z$ and let μ be a probability distribution on $X \times Y$. The (μ, ϵ)-distributional communication complexity of f, written $D^{\mu}_{cc,\epsilon}(f)$, is the number of bits communicated

by the best deterministic protocol that outputs $f(x, y)$ on at least a $1 - \epsilon$ fraction (with respect to μ) of inputs $(x, y) \in X \times Y$.

For certain distributions μ, distributional communication complexity can be much lower than randomised communication complexity; this is unsurprising as the first is an average-case model, whereas the second is a worst-case model. However, it turns out that the worst case distributional complexity over all distributions μ completely characterises public-coin randomised communication complexity.

Theorem 63. $R^{pub}_{cc}(f) = \max_\mu D^\mu_{cc,1/3}(f)$.

Proof. We will show that, for any distribution μ, $R^{pub}_{cc}(f) \geq D^\mu_{cc,1/3}(f)$. Consider any randomised protocol $\mathcal{P}$ for f. As for every pair of inputs (x, y) $\mathcal{P}$ outputs the right answer with probability at least $2/3$, averaging over any distribution μ on the inputs $\mathcal{P}$ is still correct with probability at least $2/3$, where the probability is taken both over the players' randomness and the input. Thus there must exist some choice of random coin flips such that the protocol, fixing that choice of coin flips, succeeds with probability at least $2/3$ with respect to μ.

We will not show the other direction here that $R^{pub}_{cc}(f) \leq \max_\mu D^\mu_{cc,1/3}(f)$, which can be proven using von Neumann's minimax theorem from the theory of zero-sum games. □

Theorem 63 gives a way of proving lower bounds on randomised protocols; one picks a suitable "hard" distribution μ and shows that no deterministic protocol using a small amount of communication can succeed on a large fraction of the inputs, with respect to μ. One technique for showing such a lower bound on deterministic protocols is the so-called discrepancy method.

Definition 17. *Let $f : X \times Y \to \{0, 1\}$ be a function, let R be a rectangle, and let μ be a probability distribution on $X \times Y$. Define*

$$Disc_\mu(R, f) = \left| \Pr_\mu[f(x, y) = 0 \text{ and } (x, y) \in R] - \Pr_\mu[f(x, y) = 1 \text{ and } (x, y) \in R] \right|.$$

The discrepancy of f according to μ is

$$Disc_\mu(f) = \max_R Disc_\mu(R, f),$$

where the maximum is taken over all rectangles R.

Thus, if f has low discrepancy, any rectangle in the communication matrix M^f has roughly the same number of zeros as ones (weighted by μ). Intuitively, this should mean it is hard for communication protocols to determine whether they should output 0 or 1. We will now see that this intuition is indeed correct.

Theorem 64. *Let $f : X \times Y \to \{0, 1\}$ be a function, and let μ be a probability distribution on $X \times Y$, and fix $0 < \epsilon < 1/2$. Then $D^\mu_{cc,1/2-\epsilon}(f) \geq \log_2(2\epsilon / Disc_\mu(f))$.*

Proof. Let $\mathcal{P}$ be a deterministic protocol which communicates c bits and computes f with success probability at least $1/2 + \epsilon$, with respect to μ. Then

$$\begin{aligned} 2\epsilon &\leq \Pr_\mu[\mathcal{P}(x, y) = f(x, y)] - \Pr_\mu[\mathcal{P}(x, y) \neq f(x, y)] \\ &= \sum_\ell \left(\Pr_\mu[\mathcal{P}(x, y) = f(x, y) \text{ and } (x, y) \in R_\ell] - \Pr_\mu[\mathcal{P}(x, y) \neq f(x, y) \text{ and } (x, y) \in R_\ell] \right), \end{aligned}$$

where we split the sum up in terms of the leaves ℓ of the communication protocol, and R_ℓ is the rectangle corresponding to leaf ℓ. As, for each ℓ, $\mathcal{P}(x,y)$ is constant for all $(x,y) \in R_\ell$, this is upper bounded by

$$\sum_\ell \left| \Pr_\mu[f(x,y) = 0 \text{ and } (x,y) \in R_\ell] - \Pr_\mu[f(x,y) = 1 \text{ and } (x,y) \in R_\ell] \right| = \sum_\ell \mathrm{Disc}_\mu(R_\ell, f) \leq 2^c \mathrm{Disc}_\mu(f).$$

So $2^c \mathrm{Disc}_\mu(f) \geq 2\epsilon$, which is the theorem. □

As an example application of this method, we now prove a lower bound on the *randomised* communication complexity of the inner product function.

Theorem 65. *Let $IP : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be the inner product function over $\mathbb{F}_2$: $IP(x,y) = x \cdot y = \bigoplus_{i=1}^n x_i y_i$. Then $R_{cc}^{pub}(IP) \geq n/2 - O(1)$.*

Proof. We will use Theorem 64, taking μ to be the uniform distribution. Once again, we let N be the matrix defined by $N_{xy} = 1 - 2M_{xy}^{\mathrm{IP}}$. Observe that, for any rectangle $R = A \times B$,

$$\mathrm{Disc}_\mu(R, \mathrm{IP}) = \frac{\sum_{x \in A} \sum_{y \in B} N_{xy}}{2^{2n}} = \frac{a^T N b}{2^{2n}},$$

where we write a for the characteristic vector of A, i.e. $a_x = 1 \Leftrightarrow x \in A$, and b for the characteristic vector of B. We showed in Lemma 60 that N is an orthogonal matrix (up to a normalisation factor of $2^{n/2}$). Thus

$$a^T N b \leq |a^T||Nb| = 2^{n/2}|a||b| = 2^{n/2}|A||B| \leq 2^{3n/2},$$

where $|v| = \left(\sum_i v_i^2\right)^{1/2}$ is the Euclidean norm of a vector v. Thus $\mathrm{Disc}_\mu(\mathrm{IP}) \leq 2^{-n/2}$. The claim then follows from Theorem 64. □

6.6.5 Application: Time-space tradeoffs

We now give an application of communication complexity lower bounds to the apparently unconnected topic of time-space tradeoffs for multiple-tape Turing machines. Proving lower bounds on the time complexity of explicit functions is in general a challenging task. In the case of time-space tradeoffs, we lower our sights to attempting to show that if the space used to compute some function is low, then the time used must be high.

The rough idea behind the connection to communication complexity is as follows. We split the input tape into two parts, corresponding to two distributed inputs. As the input tape is read-only, information can only be stored on the other tapes. Thus an algorithm which uses a small amount of time and space corresponds to an efficient protocol for computing a function of these distributed inputs. This idea can be formalised as the following theorem.

Theorem 66. *Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ be a function. Let M be a multiple-tape Turing machine which runs in time $T(n)$ and space $S(n)$ on inputs of size $3n$, accepts all inputs in the set*

$$\{x\, 0^n\, y : x, y \in \{0,1\}^n, f(x,y) = 1\}$$

and rejects all inputs in the set

$$\{x\, 0^n\, y : x, y \in \{0,1\}^n, f(x,y) = 0\}.$$

Then $D_{cc}(f) = O(T(n)S(n)/n)$.

Proof. Given input (x, y), in order to compute $f(x, y)$ Alice and Bob simulate the operation of the Turing machine M on input $x\,0^n\,y$, and output 1 or 0 depending on whether M accepts or rejects this input. Divide the input tape into "x", "0" and "y" regions of n bits each. At any point in time, M's head on the input tape is either in the x region (in which case Alice simulates the head's movement), in the y region (in which case Bob simulates its movement), or in the 0 region. In this case, the last player whose region the head was in continues to simulate its movement until it reaches the other player's region.

Whichever player is simulating the machine at any time knows the contents of the input tape under M's head. In order for the simulation to work, this player must also know everything else about M's current configuration. This consists of M's current state (a constant number of bits) and the contents of each of the used cells on the other tapes, together with M's head positions (in total, $O(S(n))$ bits). Whenever the simulation switches from Alice to Bob (or vice versa), this information must be communicated. As the head on the read-only tape moves at most 1 position per time step, there are at least n steps between each pair of times when the simulation switches, so there are at most $T(n)/n$ switches in total. The total amount of communication required is therefore $O(T(n)S(n)/n)$. □

Theorem 66 may seem somewhat esoteric, but it can be applied to some fairly natural problems. For example, consider the language $\mathcal{PAL} \subset \{0,1\}^*$ of *palindromes*, defined as follows:

$$\mathcal{PAL} = \{ww^R : w \in \{0,1\}^*\},$$

where w^R is the string w in reverse order. Let M be any Turing machine that decides $\mathcal{PAL}$. Defining $f(x, y) = 1$ if $x = y^R$, and 0 otherwise, M accepts all inputs in

$$\{x\,0^n\,y : x, y \in \{0,1\}^n, f(x, y) = 1\}$$

and rejects all inputs in the set

$$\{x\,0^n\,y : x, y \in \{0,1\}^n, f(x, y) = 0\}.$$

Observe that, for inputs of size n, by the log-rank lower bound $D_{cc}(f) = n$. Thus, by Theorem 66, if M operates in time $T(n)$ and space $S(n)$ on inputs of size $m = 3n$, $T(n)S(n) = \Omega(n^2)$. That is, any multiple-tape Turing machine recognising palindromes in time $T(n)$ and $S(n)$ satisfies $T(n)S(n) = \Omega(n^2)$.

The language $\mathcal{PAL}$ can be decided in linear time and linear space (by copying the first half of the input to a work tape and checking if the reversal of the second half matches the first half), or alternatively in quadratic time and logarithmic space (by checking for whether the i'th character of the input matches the $(m - i + 1)$'th character, for each i in turn). Thus the above bound is essentially tight. This bound may seem rather weak – but in fact no unconditional super-linear time lower bounds are known for computing *any* explicitly given "natural" family of functions in the multiple-tape Turing machine model! (Note that the languages occurring in the Time Hierarchy Theorem are not considered to be "natural".)

6.6.6 Extending the proof of Theorem 50 to TQBF

The idea of arithmetisation can be used to prove that in fact TQBF $\in$ IP, and hence IP = PSPACE. Given a quantified boolean formula

$$\psi = \forall x_1 \exists x_2 \forall x_3 \ldots \exists x_n \phi(x_1, \ldots, x_n),$$

where ϕ is a 3-CNF[5] boolean formula, we create a polynomial $P_\phi(X_1, \ldots, X_n)$ in the same way as before. Then ψ is true if and only if

$$\prod_{X_1 \in \{0,1\}} \coprod_{X_2 \in \{0,1\}} \prod_{X_3 \in \{0,1\}} \cdots \coprod_{X_n \in \{0,1\}} P_\phi(X_1, \ldots, X_n) = 1,$$

where we use the notation

$$\coprod_{X_n \in \{0,1\}} g(X_n) := 1 - (1 - g(0))(1 - g(1))$$

for the arithmetisation of $\exists$. We will check this using essentially the same idea as for 3-SAT$_K$, but will need to introduce a new technical trick. Imagine we want to verify that

$$\prod_{X_1 \in \{0,1\}} \coprod_{X_2 \in \{0,1\}} \prod_{X_3 \in \{0,1\}} \cdots \coprod_{X_n \in \{0,1\}} g(X_1, \ldots, X_n) = 1 \tag{6.2}$$

for some polynomial $g : \mathbb{F}_p^n \to \mathbb{F}_p$. If we write

$$h(X_1) = \coprod_{b_2 \in \{0,1\}} \prod_{b_3 \in \{0,1\}} \cdots \coprod_{b_n \in \{0,1\}} g(X_1, b_2, \ldots, b_n),$$

then (6.2) holds if and only if $h(0)h(1) = 1$, so checking this claim suffices to solve TQBF. Unfortunately, if $h(X_1)$ is defined in the same way as before (i.e. recursively), its degree could become exponentially large; each operator $\prod$ and $\coprod$ can double the degree of the polynomial. In particular, $h(X_1)$ could become a polynomial of degree $\Omega(2^n)$, which means that the prover cannot necessarily communicate it efficiently to the verifier.

However, there is a trick which can fix this. The claim (6.2) only uses values $X_i \in \{0, 1\}$, which implies that $X_i^k = X_i$ for all k. This means that we can convert g into a multilinear polynomial $\tilde{g}$ (i.e. a polynomial of degree at most 1 in each variable X_i) such that $\tilde{g}(X_1, \ldots, X_n) = g(X_1, \ldots, X_n)$ on all $X_1, \ldots, X_n \in \{0, 1\}$. Such polynomials have concise descriptions and can hence be communicated efficiently from the prover to the verifier.

For any polynomial p, let $L_i(p)$ be the polynomial defined as

$$L_i(p)(X_1, \ldots, X_n) = X_i p(X_1, \ldots, X_{i-1}, 1, X_{i+1}, \ldots, X_n) + (1 - X_i) p(X_1, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n).$$

It is easy to see that $L_i(p)$ is linear in X_i and $L_i(p)(X_1, \ldots, X_n) = p(X_1, \ldots, X_n)$ for $X_1, \ldots, X_n \in \{0, 1\}$. Thus, in particular,

$$L_1 L_2 \ldots L_n g(X_1, \ldots, X_n)$$

is multilinear for any polynomial g. So claim (6.2) is equivalent to

$$\prod_{X_1 \in \{0,1\}} L_1 \coprod_{X_2 \in \{0,1\}} L_1 L_2 \prod_{X_3 \in \{0,1\}} L_1 L_2 L_3 \cdots \coprod_{X_n \in \{0,1\}} L_1 L_2 \ldots L_n g(X_1, \ldots, X_n) = 1. \tag{6.3}$$

This expression is of size $O(n^2)$. As before, we now prove that claim (6.3) can be tested by the verifier using a recursive procedure. Assume that, for some polynomial $g(X_1, \ldots, X_k)$ and any

[5] TQBF remains PSPACE-hard under this restriction, though we technically did not show this in Theorem 32.

$b_1, \ldots, b_k, K$, the prover can convince the verifier that $g(b_1, \ldots, b_k) = K$ with probability 1 if this is true, and with probability at most ϵ if this is false. Then let $h(X_1, \ldots, X_k)$ be the polynomial obtained by taking

$$h(X_1, \ldots, X_k) = \mathcal{O}g(X_1, \ldots, X_k),$$

where $\mathcal{O}$ is any of the three operations $\coprod_{X_i \in \{0,1\}}$, $\prod_{X_i \in \{0,1\}}$, or L_i, for some i. Hence h depends on at most $k-1$ variables in the first two cases, and at most k in the third. Let d be an upper bound on the degree of h with respect to X_i; for us, $d \leq 3m$. We want to be able to check that $h(b_1, \ldots, b_k) = K$ for any $b_1, \ldots, b_k, K$; we will show that if this is not true, the prover can convince the verifier that it is true with probability at most $\epsilon + d/p$. Assuming $i = 1$ without loss of generality, we have three cases to check.

- $\mathcal{O} = \prod_{X_1 \in \{0,1\}}$. The same as in the previous protocol: the prover sends the verifier a polynomial $\tilde{h}(X_1)$ which is supposed to satisfy $\tilde{h}(X_1) = g(X_1, b_2, \ldots, b_k)$. The verifier checks whether $\tilde{h}(0)\tilde{h}(1) = K$. If not, he rejects; if so, he picks $r \in \mathbb{F}_p$ at random and asks the prover to prove $\tilde{h}(r) = g(r, b_2, \ldots, b_k)$.
- $\mathcal{O} = \coprod_{X_1 \in \{0,1\}}$. Essentially the same, but testing whether $1 - (1 - \tilde{h}(0))(1 - \tilde{h}(1)) = K$.
- $\mathcal{O} = L_1$. Once again the prover sends the verifier $\tilde{h}(X_1)$ which is supposed to satisfy $\tilde{h}(X_1) = g(X_1, b_2, \ldots, b_k)$. This time, the verifier picks $r \in \mathbb{F}_p$ at random and checks that $r\tilde{h}(1) + (1 - r)\tilde{h}(0) = K$. If not, the verifier rejects; if so, he picks $r' \in \mathbb{F}_p$ at random and asks the prover to prove that $\tilde{h}(r') = g(r', b_2, \ldots, b_k)$.

The proof of correctness is now essentially the same as the remainder of the proof of Theorem 50.

Lecture 7

Circuit Complexity

Turing machines are an (arguably!) nice model for reasoning about computation in a very general sense. However, modern computers do not usually look much like Turing machines, but instead are built from circuits of electronic components. In order to model this more directly, we consider the circuit model of computation.

For $n \in \mathbb{N}$, an n-input, single-output boolean circuit C is a directed acyclic graph which has one sink (node with out-degree 0). Sources (i.e. nodes with in-degree 0) are labelled with variables $x_1, \ldots, x_n \in \{0,1\}$, or with one of the constants 0 or 1, and are called input nodes; the sink is called the output node. Non-sources are called gates and labelled with one of $\wedge$, $\vee$ or $\neg$ (i.e. AND, OR and NOT). Nodes labelled with $\wedge$ or $\vee$ have in-degree (also called *fan-in*) 2, and nodes labelled with $\neg$ have fan-in 1. Similarly, the out-degree of a node is called its fan-out. The size of C is the number of nodes in C, and is written $|C|$.

C computes a boolean function $C : \{0,1\}^n \to \{0,1\}$ in the natural way. Each node has a value, defined recursively as follows: the value of an input node is (depending on its label) the value of its corresponding variable x_i, or the constant 0 or 1, and the value of other nodes is determined by applying the logical operation labelling that node to the values of the nodes connected to it. $C(x)$ is then defined to be the value of the output node on input x. See Figure 7.1 for an example of a circuit.

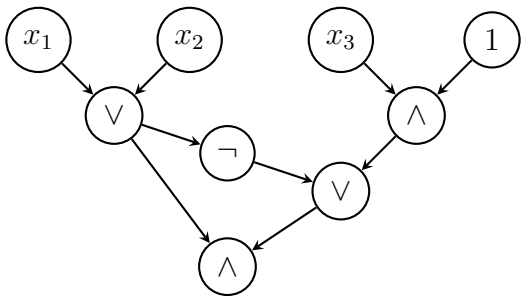


Figure 7.1: A circuit C computing a boolean function of 3 variables. One can verify that in fact $C(x) = (x_1 \vee x_2) \wedge x_3$.

Observe that every boolean formula can be written as a boolean circuit. The universality of boolean formulae (Theorem 11) thus implies that boolean circuits can compute any boolean function $f : \{0,1\}^n \to \{0,1\}$. However, the construction of Theorem 11 may use as many as $\Omega(n2^n)$ gates (there can be $\Omega(2^n)$ clauses, and each clause uses n gates). This may appear somewhat inefficient; ideally, we would like circuits of size polynomial in n. However, this is not possible for every function f, as the following theorem (which goes back to Shannon in 1949) shows:

Theorem 67. *For every integer $n > 1$, there exists a boolean function $f : \{0,1\}^n \to \{0,1\}$ which cannot be computed by any circuit of size at most $2^n/(2n)$.*

Proof. The proof is a counting argument. There are 2^{2^n} boolean functions on n bits. We will upper bound the number $C_{n,m}$ of circuits on n bits with m gates. Each such circuit is defined by choosing which type of gate each node computes, and which gates feed into each node. For each gate we have at most $(n+5)m^2$ possibilities for these, so we have at most $((n+5)m^2)^m$ different circuits in total. This is a rough over-estimate as many of these possibilities will not lead to valid circuits. We now simply observe that, for $m \leq 2^n/(2n)$,

$$\log_2 C_{n,m} \leq \frac{2^n}{2n} \log_2 \left(\left(\frac{n+5}{4n} \right) 2^{2n} \right) < 2^n = \log_2 2^{2^n},$$

valid for $n \geq 2$. □

Observe that the same proof technique even shows that all but an exponentially small fraction of functions $f : \{0,1\}^n \to \{0,1\}$ require exponential-size circuits. Nevertheless, no explicit example of a family of such functions is known!

7.1 Polynomial-size circuits

We now connect the circuit model with our previously studied notion of complexity classes.

Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-size circuit family $\mathcal{C}$ is a sequence $C_0, C_1, \ldots$ of boolean circuits, where C_n has n inputs and one output, and $|C_n| \leq T(n)$ for all n. Let $\mathcal{L} \subseteq \{0,1\}^*$ be a language. We say that a circuit family $\mathcal{C}$ decides $\mathcal{L}$ if, for every n and every $x \in \{0,1\}^n$, $x \in \mathcal{L} \Leftrightarrow C_n(x) = 1$. We further say that $\mathcal{L} \in \mathsf{SIZE}(T(n))$ if there exists a $T(n)$-size circuit family $\mathcal{C}$ such that $\mathcal{C}$ decides $\mathcal{L}$.

Observe that, by contrast with the Turing machine model, we do not demand that a single circuit works for arbitrary input sizes n. Here, we are allowed to have a completely different circuit for each n. By analogy with the Turing machine model, we will be interested in "small" classes of circuits. Let $\mathsf{P/poly}$ be the class of languages decided by polynomial-sized circuit families, i.e.

$$\mathsf{P/poly} = \bigcup_{c>0} \mathsf{SIZE}(n^c).$$

We first show that polynomial-sized circuits are at least as powerful as polynomial-time Turing machines.

Theorem 68. $\mathsf{P} \subseteq \mathsf{P/poly}$.

Proof. Consider a language $\mathcal{L} \in \mathsf{P}$ and let $x \in \{0,1\}^n$. We will show that there is a circuit C of size $\text{poly}(n)$ which computes the function $C(x) = 1 \Leftrightarrow x \in \mathcal{L}$. The proof is similar to the proof of the Cook-Levin theorem.

Let M be a Turing machine that decides $\mathcal{L}$ in time n^k, for some constant k, and let T be the tableau (defined as in the Cook-Levin theorem) describing the computational steps M takes on input x. The first row and the first and last columns of T are known in advance (being determined only by x, equal to a column of $\triangleright$'s, and equal to a column of $\square$'s, respectively). Because of the locality of Turing machines, each other cell T_{ij} depends only on three cells ($T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i+1,j-1}$) in the row above. Thus the contents of each cell T_{ij} is a function of three other cells. Let S be the set of symbols used in the tableau. Each symbol can be encoded as a fixed-length binary string (of length m, say), so the tableau can be thought of as an $n^k \times n^k \times m$ array $T_{ij\ell}$. By the above observation, each entry $T_{ij\ell}$ depends only on $3m$ other entries. This means that we can write

$$T_{ij\ell} = f_\ell(T_{i-1,j-1,1}, \ldots, T_{i-1,j-1,m}, T_{i,j-1,1}, \ldots, T_{i+1,j-1,m}),$$

where $f_\ell : \{0,1\}^{3m} \to \{0,1\}$. Each f_ℓ can thus be described by a circuit C_ℓ, which is of constant size and depends only on M. The overall circuit C consists of the repeated application of these individual circuits C_ℓ, with one application for each cell (i,j). For each entry T_{ij}, the input gates of C_ℓ are attached to the output gates of $T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i+1,j-1}$. Finally, the output of C is determined by making C output 1 if and only if the last row of the tableau corresponds to M outputting 1.

It is clear that the resulting circuit is of size $\text{poly}(n)$. Further note that (as will be important later) the above reduction can be performed in space $O(\log n)$. Each circuit C_ℓ is fixed (i.e. depends only on M). Describing C thus requires writing down the input gates (which depend only on $|x|$), generating many copies of C_ℓ and identifying correct inputs and outputs of these circuits, and writing down the output gate of C. Because of the regular structure of C, this can be done in logarithmic space. □

The above inclusion is in fact proper in a very strong sense: there even exist languages in $\mathsf{P/poly}$ which are undecidable! Let UHALT be the following language.

$$\text{UHALT} = \{1^n : \text{ the binary expansion of } n \text{ encodes a pair } (M,x) \text{ such that } M \text{ halts on input } x\}.$$

It is clear that UHALT is undecidable. On the other hand, we have the following result.

Theorem 69. *UHALT is in* $\mathsf{P/poly}$.

Proof. The theorem will follow from showing that every *unary language* is in $\mathsf{P/poly}$, where a unary language $\mathcal{L} \subset \{0,1\}^*$ is a language of the form $\mathcal{L} \subseteq \{1^n : n \in \mathbb{N}\}$. We describe a circuit family $\mathcal{C}$ for deciding $\mathcal{L}$, where each member C_n of the family behaves as follows. If $1^n \in \mathcal{L}$, then the circuit computes the AND of the n input bits (which can be done with a linear size circuit). Otherwise, the circuit always outputs 0. □

In order to conclude that UHALT $\in \mathsf{P/poly}$, it was sufficient to show that there exists a circuit to solve any given instance of UHALT, without needing to actually construct such a circuit. We would like to avoid such pathological cases in our model of circuit complexity. To do so, we introduce the notion of uniformly generated circuits.

A circuit family $\mathcal{C} = C_0, C_1, \ldots$ is said to be L-uniform if there is a logarithmic-space Turing machine which, on input 1^n, outputs a description of the circuit C_n (i.e. if there exists an implicitly logspace-computable function which maps 1^n to C_n). This type of circuit turns out to be precisely equivalent to P.

Theorem 70. *A language $\mathcal{L} \subseteq \{0,1\}^*$ can be decided by an L-uniform circuit family if and only if $\mathcal{L} \in \mathsf{P}$.*

Proof. ($\Rightarrow$): If $\mathcal{L} \subseteq \{0,1\}^*$ is decided by an L-uniform circuit family $\mathcal{C}$, then for any $x \in \{0,1\}^*$, we can determine whether $x \in \mathcal{L}$ using a polynomial-time Turing machine by first constructing the required circuit $C_{|x|}$, then evaluating it.

($\Leftarrow$): Follows from the proof of Theorem 68, which gives a logarithmic-space construction of a circuit $C_{|x|}$ which outputs 1 if and only if $x \in \mathcal{L}$. □

The same idea allows us to give a natural complete problem for P. As with the class NL (Section 4.3), to formulate a sensible notion of complete problems for P it is necessary to use a weaker notion of reductions than for NP (which motivates the above introduction of the idea of L-uniformity). As with NL, the notion of reductions we use is implicit logarithmic space reductions. We say that a language $\mathcal{A}$ is P-complete if $\mathcal{A} \in \mathsf{P}$, and for all $\mathcal{B} \in \mathsf{P}$, $\mathcal{B} \leq_{\mathsf{L}} \mathcal{A}$. P-complete problems are the "hardest" problems in P; indeed, if $\mathcal{L}$ is P-complete and $\mathcal{L} \in \mathsf{L}$, then $\mathsf{P} = \mathsf{L}$.

The Circuit Value problem is defined as follows. The input is a pair (C, x), where C is the description of an n-input circuit and $x \in \{0,1\}^n$. The problem is to decide whether $C(x) = 1$.

Theorem 71. Circuit Value *is P-complete.*

Proof. Evaluating $C(x)$ can clearly be done in polynomial time; for the other direction, Theorem 68 gives an implicitly logspace-computable reduction from any problem in P to Circuit Value. □

Similarly, we can give a characterisation of the class NP in terms of circuits. The Circuit SAT problem is defined as follows: the input is a description of an n-input circuit C, and the problem is to decide whether there exists $x \in \{0,1\}^n$ such that $C(x) = 1$. As boolean formulae are special cases of circuits, Circuit SAT is at least as hard as 3-SAT. We now give an alternative proof of this fact which follows directly from the definition of NP.

Theorem 72. Circuit *SAT is NP-complete.*

Proof. Circuit SAT is clearly in NP, as given $x \in \{0,1\}^n$, we can evaluate $C(x)$ in polynomial time. For the other direction, if $\mathcal{L} \in \mathsf{NP}$ then there is a polynomial-time Turing machine M and a polynomial p such that $x \in \mathcal{L}$ if and only if $M(x, w) = 1$ for some $w \in \{0,1\}^{p(n)}$.

The proof of Theorem 68 shows that, given the pair $(\underline{M}, x)$, we can write down a circuit C_x in polynomial time such that $C_x(w) = M(x, w)$ for all $w \in \{0,1\}^{p(n)}$. Thus solving CIRCUIT SAT for C_x allows us to determine whether $x \in \mathcal{L}$. □

This now allows us to go full circle and prove NP-hardness of 3-SAT using CIRCUIT SAT – i.e. to give an alternative proof of the Cook-Levin theorem.

Theorem 73. CIRCUIT *SAT* $\leq_{\mathsf{P}}$ *3-SAT.*

Proof sketch. Given a circuit C containing AND, OR and NOT gates, we produce a 3-CNF formula ϕ as follows. Each node v_i has a corresponding variable z_i in ϕ. We then add clauses to ϕ such that the input and output constraints of each gate are satisfied. For example, if $v_i = v_j \wedge v_k$ is an AND gate, we want to add the condition $(z_i = z_j \wedge z_k)$, which can easily be written as a CNF formula; the same applies for OR and NOT. Finally, if the output node is v_o, we add a clause (z_o) to specify that the output should be 1. It is easy to see that ϕ is satisfiable if and only if C is satisfiable. □

As the reductions occuring in these two theorems are parsimonious, this finally proves the claim made in Section 8.1 that the counting problem #3-SAT is #P-complete.

7.2 Restricted depth circuits

We have seen that size in the model of uniformly generated circuits is roughly equivalent to time in the world of Turing machines. However, the circuit model allows us to consider other measures of complexity which do not necessarily correspond to natural parameters in the Turing machine model. A particularly interesting such measure is circuit depth, where the depth of a circuit C is the maximal number of nodes on a path from an input node to the output node. We now define two classes of depth-limited computations.

- For every d, a language $\mathcal{L} \subseteq \{0,1\}^*$ is in the class NC^d if $\mathcal{L}$ can be decided by an L-uniform family of circuits $\{C_n\}$ such that each C_n has size $\mathrm{poly}(n)$ and depth $O(\log^d n)$. We write $\mathsf{NC} = \bigcup_{d \geq 0} \mathsf{NC}^d$.
- For every d, the class AC^d is defined similarly, but with the extension that we allow gates in each circuit C_n to have unbounded fan-in. That is, each OR and AND gate can be applied to arbitrarily many input bits. We write $\mathsf{AC} = \bigcup_{d \geq 0} \mathsf{AC}^d$.

NC stands for "Nick's Class" in honour of Nick Pippenger, who first defined the class, while the A in AC stands for "alternations". The discussion of the circuit model given here is largely based on Arora-Barak (chapter 6) and Papadimitriou (chapters 11 and 15). Thus, for each d, the class AC^d is at least as powerful as NC^d. Indeed, we have

$$\mathsf{NC}^d \subseteq \mathsf{AC}^d \subseteq \mathsf{NC}^{d+1}$$

(you will show the second inclusion in an exercise). The class NC^0 is very limited as the output of any circuit in this class can depend only on a constant number of input bits; AC^0

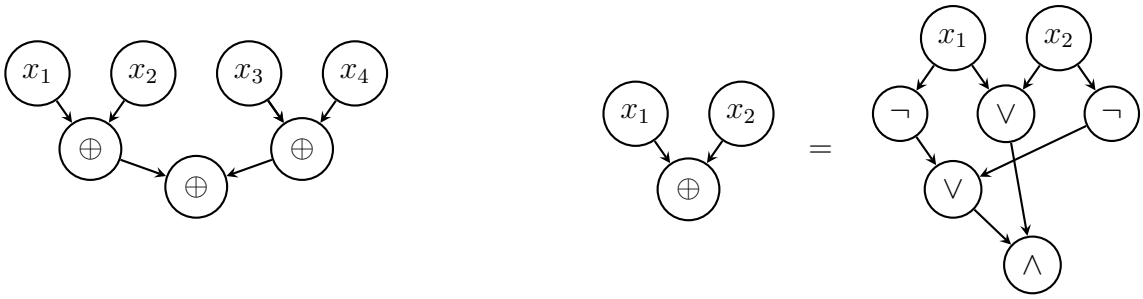


Figure 7.2: Recursive construction for computing PARITY, and this construction mapped onto the standard circuit model.

does not have this restriction and is arguably the simplest interesting class of circuits which one could consider.

There are two main motivations for studying such classes. One motativation is their connection to reality; the class NC in particular corresponds quite well to a model of massively parallel computing where we have access to a large number of processors with efficient communication between them (such a model was not realistic for some time after NC was first studied, but nowadays is not too far from the truth). Another motivation is the fact that it seems difficult to prove a super-polynomial lower bound on circuit size for an explicit family of functions. Restricting ourselves to considering circuits in NC or AC might allow better lower bounds to be proven.

In particular, we will consider the problem PARITY, which is defined as follows: given $x \in \{0,1\}^n$, does x contain an odd number of 1's? Observe that this problem is in NC^1: given an n-bit string x, solving PARITY is the same as computing $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, which can be done in depth $O(\log n)$ by a recursive construction illustrated in Figure 7.2.

However, we have the following result.

Theorem 74. PARITY $\notin \mathsf{AC}^0$. *Hence* $\mathsf{AC}^0 \neq \mathsf{NC}^1$.

The class AC^0 appears very weak, so it is not clear that this is an interesting result. However, it is almost the strongest lower bound known on circuit size!

The basic idea behind Theorem 74 is as follows. We will show that any function $f : \{0,1\}^n \to \{0,1\}$ computed by a constant-depth, polynomial-size circuit has the property that we can force it to become constant by fixing strictly less than n of its input variables. As PARITY does not have this property, PARITY $\notin \mathsf{AC}^0$. To gain some intuition, consider the special case where f is computed by a k-CNF formula for $k < n$ (recall that a k-CNF (resp. k-DNF) formula is a boolean formula which is an AND of ORs (resp. OR of ANDs) where each OR (resp. AND) involves at most k variables). We can make such a formula constant by forcing one clause to evaluate to 0 or 1, respectively, which requires fixing at most k of the variables. Thus PARITY cannot be computed by such a formula.

The proof of Theorem 74 will be based on extending this idea to general circuits, using the following "switching lemma", which we state without proof. The lemma uses the concept

of random restrictions of a boolean formula. If f is a boolean function and ρ is a partial assignment (known as a restriction) to the variables of f, let $f|_\rho$ denote the function obtained by fixing the variables of ρ.

Lemma 75 (Håstad's Switching Lemma)**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be expressible as a k-DNF, and let $f|_\rho$ be a random restriction that assigns random values to t randomly selected input bits. Then, for every $s \geq 2$,*

$$\Pr_\rho[f|_\rho \text{ is not expressible as } s\text{-CNF}] \leq \left((1-t/n)k^{10}\right)^{s/2}.$$

Arora-Barak chapter 6 includes a proof of this lemma.

By symmetry (i.e. applying Lemma 75 to $\neg f$), we get the same result with the terms DNF and CNF interchanged.

Proof of Theorem 74 (sketch). We assume that we are given an AC^0 circuit C of depth d in the following form (any circuit can be converted into such a form).

- The circuit is a tree (i.e. all gates have fan-out 1);
- All $\neg$ gates are at the input of the circuit;
- At each level of the tree, all gates are either $\vee$ or $\wedge$;
- The first level after the input consists of $\vee$ gates of fan-in 1.

Let n^c be an upper bound on the number of gates in C. We randomly restrict more and more variables; the idea is that each restriction reduces the depth by 1 while retaining constant fan-in at the first level of the circuit. Let n_i be the number of unrestricted variables after step i. At step $i+1$ we randomly restrict $\lceil n_i - \sqrt{n_i} \rceil$ variables. As $n_0 = n$, $n_i \approx n^{2^{-i}}$. Set $k_i = 10c2^i$. We aim to show by induction that, with high probability, after restriction i we have a circuit of depth at most $(d-i)$ and with fan-in at most k_i at the first level.

Suppose the first level contains $\vee$ gates, so the second level contains $\wedge$ gates. Assuming that the function computed by each $\wedge$ gate is a k_i-CNF, by Lemma 75, after the $(i+1)$'th step it will be expressible as a k_{i+1}-DNF with probability at least

$$1 - \left(k_i^{10} n^{-2^{-i-1}}\right)^{k_{i+1}/2},$$

which is at least $1 - (10n^c)^{-1}$ for large enough n (observing that k_i is constant with respect to n!). As the bottom gate of a DNF formula is $\vee$, we can merge the gates at the second level with the level below, decreasing the circuit's depth by 1. On the other hand, if the first level contained $\wedge$ gates, we can transform the k_i-DNF of the next level into a k_{i+1}-CNF using the same idea.

We repeat this process $d-2$ times. With constant probability we end up with a depth 2 circuit with fan-in at most k_{d-2} at the top level. To see this, note that we apply Lemma 75 once per gate, so using a union bound the probability that all of the applications were

successful is at least $1-n^c(10n^c)^{-1} = 9/10$. A depth 2 circuit is just a k_{d-2}-CNF or k_{d-2}-DNF formula. But if we fix at most $k_{d-2} = O(1)$ of the variables in such a formula, we can make the formula constant (by fixing one clause to 0 or 1, respectively). The PARITY function is not constant under any restriction of fewer than n variables, so PARITY $\notin \mathsf{AC}^0$. □

Self Test Question:
Prove that $\mathsf{AC}^d \subseteq \mathsf{NC}^{d+1}$ for any $d \geq 0$.

Solution:
Given an AND (or OR) gate applied to k bits, we can replace this with a tree construction that uses only gates of fan-in two, in the same way as was done for PARITY. This construction has depth $O(\log k)$. As AC^d circuits have size at most $\text{poly}(n)$, we never need to apply a gate to more than $\text{poly}(n)$ input bits. Hence the depth of the circuit increases by at most a multiplicative factor of $O(\log n)$.

Self Test Question:
Prove that $\mathsf{NC}^1 \subseteq \mathsf{L}$.

Solution:
The idea is similar to the proof that TQBF is in PSPACE. A circuit is a directed acyclic graph and can be evaluated recursively as follows. For each node v, starting with the output node, evaluate each of the nodes pointing to v in turn, reusing space. We need to store the path from the output node to the node which we are currently evaluating, and the outputs of nodes evaluated so far. For a circuit of depth D, this requires $O(D)$ space; in particular, for a circuit of depth $O(\log n)$, this is $O(\log n)$ space.

Self Test Question:
Given two $n \times n$ boolean matrices A and B (i.e. matrices which contain only 0's and 1's), define the boolean matrix product

$$(A \circ B)_{ij} = \bigvee_{k=1}^{n} (A_{ik} \wedge B_{kj}).$$

Also define the problem "BMM" as follows: given two $n \times n$ boolean matrices A and B, and integers s and t such that $1 \leq s, t \leq n$, determine whether $(A \circ B)_{st} = 1$.

1. Show that BMM $\in \mathsf{AC}^0$.
2. Show that, given an $n \times n$ boolean matrix A, and any integer $k = O(\log n)$, the problem of determining whether $(A^{\circ 2^k})_{st} = 1$ is in AC^1. Here $A^{\circ 2^k}$ is the 2^k-fold product $A \circ A \circ \cdots \circ A$.
3. Conclude that $\mathsf{NL} \subseteq \mathsf{AC}^1$.

Solution:

1. We can write down a constant-depth circuit containing n^3 gates G_{ijk} such that $G_{ijk} = A_{ik} \wedge B_{kj}$, and a set of n^2 gates G'_{ij} such that $G'_{ij} = \bigvee_{k=1}^{n} G_{ijk}$. We still need to extract the correct bit G'_{st}. But this can be done by having a set of n^2 gates V_{ij} such that V_{ij} evaluates to true if and only if $s = i$, $t = j$. Each such gate can be produced by taking the AND of the $O(\log n)$ bits making up s and t. Then it suffices to take the AND of V_{ij} and G'_{ij} for all i, j, and take the OR of the resulting gates.

2. For any A, by the solution to the previous question we can compute every bit of A^2 in constant depth. Thus, for any integer k, we can compute any desired entry of $A^{\circ 2^k}$ in AC^1 with a tree construction.

3. It suffices to show that $\textsc{Path} \in \mathsf{AC}^1$. Given a directed graph G on n nodes with adjacency matrix A, there is a path from s to t precisely when $((A + I)^{\circ m})_{st} = 1$, for some $m \geq n$. Thus, taking m to be the smallest power of 2 larger than n and using the algorithm of part (b), $\textsc{Path} \in \mathsf{AC}^1$.

7.3 Circuits and randomised algorithms

Recall that it is not known whether $\mathsf{BPP} = \mathsf{P}$. However, in the circuit model one can indeed prove a result of this nature.

Theorem 76. $\mathsf{BPP} \subset \mathsf{P/poly}$.

Proof. Let $\mathcal{L} \in \mathsf{BPP}$. From the proof of Lemma 41, this implies that there exists a polynomial-time Turing machine M which, on inputs of size n, uses some random bits $r \in \{0,1\}^m$, for some $m \geq 0$, and for all $x \in \{0,1\}^n$, $\Pr_r[M(x,r) \neq f_{\mathcal{L}}(x)] \leq 2^{-n-1}$. (Recall that $f_{\mathcal{L}}(x) = 1$ if $x \in \mathcal{L}$, and $f_{\mathcal{L}}(x) = 0$ if $x \notin \mathcal{L}$.) Call a string $r \in \{0,1\}^m$ bad for x if $M(x,r) \neq f_{\mathcal{L}}(x)$, and otherwise call r good for x. For each x, at most 2^{m-n-1} strings r are bad for x. Thus, summing over all x, there are at most 2^{m-1} strings which are bad for *some* x. In particular, for each n there is a string $g \in \{0,1\}^m$ which is good for every $x \in \{0,1\}^n$. We can fix this string g as input to M and hence obtain a circuit C_n whose size is polynomial in the time used by M, and such that $C(x) = f_{\mathcal{L}}(x)$ for all $x \in \{0,1\}^n$. □

Of course, it cannot be the case that in fact $\mathsf{BPP} = \mathsf{P/poly}$, as $\mathsf{P/poly}$ contains undecidable problems. Theorem 76 can be seen as evidence that $\mathsf{BPP} = \mathsf{P}$. Additional evidence is provided by the following result, which we state informally: if there exists a language $\mathcal{L} \in \mathsf{DTIME}(2^{O(n)})$ such that deciding $\mathcal{L}$ requires exponentially large circuits, then $\mathsf{BPP} = \mathsf{P}$. See Arora-Barak chapter 20 for a discussion and proof of this claim.

7.4 Supplementary Notes

In this set of supplementary notes we discuss natural proofs and the P vs. NP problem. The notion of "natural proof" was invented by Razborov and Rudich in 1994, and has turned out to be a remarkably wide-ranging conceptual roadblock to proving $\mathsf{P} \neq \mathsf{NP}$. See Arora-Barak chapter 23 for a discussion.

As $\mathsf{P} \subseteq \mathsf{P/poly}$, one plausible way to prove $\mathsf{P} \neq \mathsf{NP}$ is to prove $\mathsf{NP} \not\subset \mathsf{P/poly}$ (again, we cannot have $\mathsf{P/poly} = \mathsf{NP}$ because $\mathsf{P/poly}$ contains undecidable problems). This could be a hopeful strategy because circuits may seem easier (or more "concrete") to reason about than Turing machines. The most direct way to prove this separation would be to show that some problem believed not to be in P (such as the Integer Factorisation problem, or any NP-complete problem) requires circuits of super-polynomial size. Indeed, the result that Parity $\notin \mathsf{AC}^0$ seems to represent progress in this direction.

Ideally, we could imagine that we would develop a test which we could apply to the truth table of any boolean function $f : \{0,1\}^n \to \{0,1\}$, which would, in time polynomial in the size of the truth table, output either "easy" (f has a circuit of size at most n^k, for some k) or "hard" (f does not have such a circuit). If we could then find a family of functions f known to be in NP, but for which this test output "hard", we would have shown $\mathsf{NP} \not\subset \mathsf{P/poly}$. As well as helping to resolve complexity-theoretic issues, such a test would be very useful in practice.

Unfortunately, it has been shown that (modulo some reasonable assumptions) such a test cannot exist, significantly restricting the potential for this approach to prove $\mathsf{P} \neq \mathsf{NP}$. The proof of this result is of some conceptual interest in itself as it links two apparently (?) disparate areas: hardness and pseudorandomness.

In order to state the result more formally, we will need some definitions. Let F_n be the set of boolean functions $f : \{0,1\}^n \to \{0,1\}$. A *property* P_n is a function $P_n : F_n \to \{0,1\}$. We say that P_n is *natural* if it satisfies the following two properties.

- **(Constructivity)** For any $f : \{0,1\}^n \to \{0,1\}$, computing $P_n(f)$ can be done in time polynomial in the size of the truth table of f, i.e. in time $2^{O(n)}$.
- **(Largeness)** At least a $2^{-O(n)}$ fraction of functions $f \in F_n$ satisfy $P_n(f) = 1$.

A good example of a natural property is the property "f does not become constant after fixing all but n^ϵ variables of the input, for some constant ϵ", which was used in the proof that Parity $\notin \mathsf{AC}^0$. It can be verified that this property is satisfied by random functions with high probability, and can be checked in time $2^{O(n)}$.

We say that P_n is *useful* against $\mathsf{P/poly}$ if the circuit size of any sequence of functions $f_1, f_2, \ldots$, where $f_n \in P_n$, is super-polynomial in n; i.e. for any constant k, for sufficiently large n, the circuit size of f_n is larger than n^k. If a natural property P_n exists which is useful against $\mathsf{P/poly}$, we therefore have an algorithm which we can apply to boolean functions to certify that they do not have small circuits. Equivalent definitions can be formulated for proofs useful against other circuit complexity classes, but we will only consider $\mathsf{P/poly}$ here.

We will also need the concept of one-way functions, which are functions which are easy to compute but hard to invert. The field of cryptography is essentially based around the idea of one-way functions (although we do not know if any such functions exist!).

A polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be a *subexponentially strong one-way function* if there is an $\epsilon > 0$ such that, for every probabilistic algorithm $\mathcal{A}$ running

in time 2^{n^ϵ}, there is a function $\delta : \mathbb{N} \to [0,1]$ such that, for every n,

$$\Pr_{x \in \{0,1\}^n}[\mathcal{A}(f(x)) = y \text{ for some } y \text{ such that } f(y) = f(x)] < \delta(n),$$

and $\delta(n) < n^{-c}$ for every constant c and sufficiently large n. In other words, any probabilistic algorithm running in time 2^{n^ϵ} has super-polynomially small probability of finding some y such that $f(y) = f(x)$. We think of f as encoding its input x in such a way that it is hard for any adversary to decode it. It is not obvious that one-way functions should exist; indeed, if they do exist, then $\mathsf{P} \neq \mathsf{NP}$. However, a number of natural functions are conjectured to be one-way. A simple example is integer multiplication: inverting the function f which treats its input as two n-bit numbers and outputs their product is essentially the Integer Factorisation problem. (Recall that the best known algorithm for this problem uses time $2^{O(n^{1/3}(\log n)^{2/3})}$.)

We are finally ready to state the main result about natural proofs.

Theorem 77. *If there exists a natural property P_n which is useful against P/poly, subexponentially strong one-way functions cannot exist.*

We prove this theorem by appealing to a result connecting pseudo-random function generators and one-way functions, which we will state without proof.

A *pseudo-random function generator* is a polynomial-time computable function $f : \{0,1\}^{n^c} \to F_n$, for some constant $c > 1$. f takes as input an n^c-bit "key" s, and outputs a function $f(s) \in F_n$ which is indistinguishable from a random function to any probabilistic algorithm $\mathcal{A}$ that runs in time polynomial in the truth table of $f(s)$, i.e. in time $2^{O(n)}$. Formally,

$$\left| \Pr_{g \in F_n}[\mathcal{A}(g) = 1] - \Pr_{s \in \{0,1\}^{n^c}}[\mathcal{A}(f(s)) = 1] \right| \leq 2^{-O(n)}.$$

In particular, observe that $\mathcal{A}$ cannot just attempt to guess the key s, as this would require time $2^{\Theta(n^c)}$. The following lemma, which we will not prove here, states that pseudo-random function generators can be constructed from one-way functions.

Lemma 78. *Given that a family of subexponentially strong one-way functions exists, then a family of pseudo-random function generators exists.*

There is a vast literature concerning pseudorandomness and one-way functions. For details of the results sketched out here, a good source is the two-volume work "Foundations of Cryptography" by Oded Goldreich[1].

Based on this lemma, we can show that the existence of useful natural proofs implies the nonexistence of pseudo-random function generators, thus proving Theorem 77.

Lemma 79. *If there exists a natural property P_n which is useful against P/poly, pseudo-random function generators cannot exist.*

Proof. Assume that such a property P_n does exist. Then we can distinguish the output of such pseudo-random function generators from true randomness, i.e.:

$$\left| \Pr_{g \in F_n}[P_n(g) = 1] - \Pr_{s \in \{0,1\}^{n^c}}[P_n(f(s)) = 1] \right| \geq 2^{-O(n)}.$$

[1] `http://www.wisdom.weizmann.ac.il/~oded/foc-drafts.html`

To see this, note that $f(s)$ is polynomial-time computable, and thus cannot satisfy the property P_n. However, by the largeness property, at least a $2^{-O(n)}$ fraction of functions will be in P_n, and hence we are able to distinguish the two cases. Note that we have used all the attributes of a natural property useful against P/poly: constructivity, largeness and usefulness. □

7.5 Advanced Technical Notes

Self Test Question:
Let 1UN-SAT be the special case of SAT where every clause contains at most one un-negated variable (for example, $x_1 \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2)$ is a valid 1UN-SAT instance). Prove that 1UN-SAT is P-complete. [HARD]

Solution:
We first show that 1UN-SAT (which is in fact usually known as Horn-SAT) is in P. Given a 1UN-SAT instance ϕ, our polynomial-time algorithm proceeds as follows. For each variable x_i which appears in a clause only containing one variable, we replace x_i with 0 or 1 as appropriate to preserve satisfiability of ϕ, thus removing x_i from any clauses containing it (which may result in producing more clauses containing only one variable). We repeat this procedure until we reach a contradiction (i.e. the formula is unsatisfiable) or are left with a formula containing only clauses with more than one variable. Each such clause must contain at least one negated variable, so we can satisfy the formula by assigning 0 to all remaining variables.
We now complete the proof that 1UN-SAT is P-complete by giving a reduction from CIRCUIT VALUE to 1UN-SAT. Given a circuit C which we want to evaluate on input $x \in \{0,1\}^n$, we represent each gate (or input) node g_i by two boolean variables p_i, n_i, with the intention being that $p_i = 1$ if g_i evaluates to true, and $n_i = 1$ if g_i evaluates to false. We now produce a formula ϕ with the following clauses:

- If g_i is an input node, and the corresponding variable is set to true, we have the clause p_i; otherwise, we have the clause n_i.
- If $g_i = \neg g_j$ is a NOT gate, we have the clauses (which we write using $\Rightarrow$ for clarity; it should be clear that these can be written as clauses which fit the definition of 1UN-SAT)

$$(p_j \Rightarrow n_i) \wedge (n_j \Rightarrow p_i).$$

- If $g_i = g_j \vee g_k$ is an OR gate, we have the clauses

$$(p_j \Rightarrow p_i) \wedge (p_k \Rightarrow p_i) \wedge ((n_j \wedge n_k) \Rightarrow n_i).$$

- If $g_i = g_j \vee g_k$ is an AND gate, we have the clauses

$$(n_j \Rightarrow n_i) \wedge (n_k \Rightarrow n_i) \wedge ((p_j \wedge p_k) \Rightarrow p_i).$$

If $C(x) = 1$, it is possible to satisfy all of the clauses by setting p_i, n_i appropriately, according to whether gate g_i evaluates to true or false. On the other hand, one can show by induction (we omit the details...) that if there is a satisfying assigment to ϕ, $C(x) = 1$. Observe that not every such satisfying assignment corresponds to a "real" assignment of truth values to gates. For example, we can satisfy the "OR gate" constraint that $g_i = g_j \vee g_k$ by an assignment that includes $p_i = n_i = p_j = 1$ (so g_i is assigned true and false simultaneously). However, the existence of any such satisfying assignment does imply existence of a satisfying assignment that does correspond to a real assignment of truth values to gates.
It remains to show that this reduction is implicitly log-space computable. But this is essentially immediate because of the regular structure of the reduction where each gate corresponds to a clause.

Lecture 8

Advanced Notes

In this chapter we gives notes on aspects of Complexity Theory which we are unable to touch on in this course. The notes might form the basis of a further Year 3 or 4 Complexity Theory course, that we might put on in future years. The notes are provided here for the interested student. As we explained at the start of these notes you are meant to be reading around the subjects you study at University. These notes are provided to help you start doing this.

8.1 Counting complexity and the class #P

Some of the complexity classes we have seen so far have nice definitions in terms of computational paths of NDTMs. In particular, we can write down the following characterisations.

- A language $\mathcal{L} \in \mathsf{P}$ if there exists a polynomial-time NDTM M such that, on input x:
  1. For all $x \in \mathcal{L}$, all of M's computational paths accept;
  2. For all $x \notin \mathcal{L}$, none of M's computational paths accept.
- A language $\mathcal{L} \in \mathsf{NP}$ if there exists a polynomial-time NDTM M such that, on input x:
  1. For all $x \in \mathcal{L}$, at least one of M's computational paths accepts;
  2. For all $x \notin \mathcal{L}$, none of M's computational paths accept.
- A language $\mathcal{L} \in \mathsf{co\text{-}NP}$ if there exists a polynomial-time NDTM M such that, on input x:
  1. For all $x \in \mathcal{L}$, all of M's computational paths accept;
  2. For all $x \notin \mathcal{L}$, at least one of M's computational paths does not accept.
- A language $\mathcal{L} \in \mathsf{BPP}$ if there exists a polynomial-time NDTM M such that, on input x:
  1. For all $x \in \mathcal{L}$, at least $2/3$ of M's computational paths accept;

2. For all $x \notin \mathcal{L}$, at most $1/3$ of M's computational paths accept.

- A language $\mathcal{L} \in \mathsf{PP}$ if there exists a polynomial-time NDTM M such that, on input x:

  1. For all $x \in \mathcal{L}$, $> 1/2$ of M's computational paths accept;
  2. For all $x \notin \mathcal{L}$, $\leq 1/2$ of M's computational paths accept.

A problem at least as difficult as all of the above complexity classes is calculating the number of accepting paths of an NDTM M on input x. This motivates the introduction of the complexity class #P (pronounced "sharp P"), which is defined as the class of all functions f such that $f(x)$ is equal to the number of accepting paths of some NDTM M on input x. Note that this is a class of functional problems, rather than decision problems. Once again, we can equivalently define this class in terms of certificates, as follows:

Definition 18. *A function $f : \{0,1\}^* \to \mathbb{N}$ is in #P if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that, for all $x \in \{0,1\}^*$,*

$$f(x) = \left|\{y \in \{0,1\}^{p(|x|)} : M(x,y) = 1\}\right|.$$

Informally speaking, #P is the class of problems which involve determining the size of a set (which is possibly exponentially large), such that membership in the set can be checked efficiently. It is perhaps easiest to understand this class by considering some simple examples of problems in #P.

The complexity class #P was introduced by Leslie Valiant in 1979, in the same paper in which he proved that computing the permanent is #P-complete, see Theorem 82 below. He received the 2010 Turing Award, the highest award in computer science, partly for this work (but also for much more!). Arora-Barak (chapter 17) and Papadimitriou (chapter 18) have nice discussions on #P-hardness and counting problems.

- #SAT: given a boolean formula ϕ, calculate the number of satisfying assignments of ϕ. This is clearly at least as hard as SAT and Tautology.

- #Path: given a graph G and two vertices s and t, count the number of simple paths from s to t (a path is called simple if it does not visit any vertex twice).

As with NP, #P has a notion of complete problems, which are the "hardest" problems in #P. Recalling that FP is the class of polynomial-time computable functions, we define FP^f (for some function $f : \{0,1\}^* \to \{0,1\}^*$) to be the class of functions which are computable in polynomial time, given access to an oracle for f. Generalising the definition of oracles for decision problems, we say that a Turing machine M has an oracle for f if it has an oracle for the language $\mathcal{L}_f = \{(x,i) : f(x)_i = 1\}$. In other words, M can extract arbitrary bits of $f(x)$ "for free".

Definition 19. *f is #P-complete if $f \in$ #P and every function $g \in$ #P is in FP^f. Thus, if f is #P-complete and $f \in$ FP, FP = #P.*

Perhaps unsurprisingly, we have the following result.

Theorem 80. *#SAT is #P-complete.*

Proof. Imagine that, given the description of some NDTM M, we could output a boolean formula ϕ such that the number of satisfying assignments of ϕ is equal to the number of accepting paths of M. If this were the case, the theorem would be proven, as given an oracle for #SAT we could compute the number of accepting paths of M, for any NDTM M. Inspecting the proof of the Cook-Levin Theorem (Section 2.2) shows that this almost gives us such a reduction. All that is required is to constrain M to accept only after a fixed number of steps, and in a fixed configuration; both of these modifications are straightforward. □

For languages $\mathcal{A}, \mathcal{B} \in \mathsf{NP}$, we say that a reduction from language $\mathcal{A}$ to $\mathcal{B}$ is *parsimonious* if it preserves the number of accepting paths, i.e. the number of solutions. The above theorem gives a parsimonious reduction from any language in NP to SAT. In fact, there also exists a parsimonious reduction from any language in NP to 3-SAT. We have technically not proven this yet, as the reduction from SAT to 3-SAT we previously gave is not parsimonious; we defer the proof of this claim until later, when we will give an alternative proof of the Cook-Levin theorem.

In order to prove that some counting problem in #P with underlying language $\mathcal{L} \in \mathsf{NP}$ is #P-complete, it therefore suffices to give a parsimonious reduction from 3-SAT to $\mathcal{L}$. More generally, it suffices to give a reduction which is parsimonious up to some easily computable additive or multiplicative constant; we also call reductions of this form parsimonious.

Although it is arguably unexciting that the counting variant of an NP-complete problem such as 3-SAT should be hard, it is more surprising that there exist #P-complete counting problems whose decision variants are in P. A simple such example is the problem of counting satisfying assignments to a *monotone* boolean formula, which is a boolean formula with no negations. For example,

$$\phi = (x_1 \wedge (x_2 \vee x_3 \vee x_4)) \vee (x_1 \vee x_3)$$

is a monotone boolean formula. Such formulae always have a satisfying assignment where all variables are set to true, so determining whether a monotone boolean formula has a satisfying assignment is clearly in P. However, we have the following result.

Theorem 81. *The problem #MON of counting the number of satisfying assignments to a monotone formula is #P-complete.*

Proof. #MON is clearly in #P. To prove #P-hardness, we give a reduction from #SAT to #MON. Given a boolean formula $\phi(x_1, \ldots, x_n)$, we produce a new formula $\psi(x_1, \ldots, x_n, y_1, \ldots, y_n)$ where ψ is the same as ϕ, but with each occurrence of $\neg x_i$ replaced with y_i, for all i. Write $\#\phi$ for the number of satisfying assignments of ϕ, for any boolean formula ϕ. Then

$$\#\phi = \# \left(\psi \wedge \left(\bigwedge_{i=1}^{n} (x_i \vee y_i) \right) \wedge \left(\bigwedge_{i=1}^{n} \neg (x_i \wedge y_i) \right) \right);$$

the number of satisfying assignments of ϕ is the same as the number of satisfying assignments of ψ where $x_i \neq y_i$ for all i. Now write

$$\psi_A = \psi \wedge \left(\bigwedge_{i=1}^{n} (x_i \vee y_i) \right), \quad \psi_B = \left(\bigvee_{i=1}^{n} (x_i \wedge y_i) \right).$$

Then (using De Morgan's law) $\#\phi = \#(\psi_A \wedge \neg\psi_B)$, where ψ_A and ψ_B are both monotone formulae. But

$$\#(\psi_A \wedge \neg\psi_B) = \#\psi_A - \#(\psi_A \wedge \psi_B),$$

simply by counting the number of assignments that satisfy ψ_A but not ψ_B. Hence $\#\phi$ can be computed as the difference between the numbers of satisfying assignments of two monotone formulae. □

Many important combinatorial problems also turn out to be #P-complete. An example, which we state without proof, is a problem from statistical physics. The *Ising model* was originally introduced as a model for studying ferromagnetism. An instance of the model is a set of n sites, each pair of which has an interaction energy V_{ij}. A configuration is an assignment $\sigma \in \{\pm 1\}^n$, where σ_i is the spin assigned to the i'th site. The energy of a configuration is given by

$$H(\sigma) = -\sum_{i<j=1}^{n} V_{ij}\sigma_i\sigma_j - B\sum_{k=1}^{n} \sigma_k,$$

where B is some fixed magnetic field intensity. The central problem associated with the Ising model is to compute the partition function

$$Z = \sum_{\sigma \in \{\pm 1\}^n} e^{-\beta H(\sigma)}$$

for some β. Observe that Z is a sum over exponentially many terms, each of which can easily be computed. It turns out that computing Z is in fact #P-hard.

8.1.1 The permanent is #P-complete

We now give a natural mathematical problem which is also #P-complete. Let M be an $n \times n$ matrix of integers. We say that M is a 0-1 matrix if it only contains 0's and 1's. The determinant of M can be defined as

$$\det(M) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} M_{i,\sigma(i)},$$

where the sum is over all permutations σ of $\{1, \ldots, n\}$, and $\operatorname{sgn}(\sigma)$ is the sign of σ, i.e. $\operatorname{sgn}(\sigma) = 1$ if σ can be written as a product of an even number of transpositions, and

$\mathrm{sgn}(\sigma) = -1$ otherwise. Of course, $\det(M)$ can be computed in polynomial time (by Gaussian elimination, for example). But consider the superficially similar quantity

$$\mathrm{perm}(M) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} M_{i,\sigma(i)},$$

which is called the permanent of M. Then we have the following result.

Theorem 82 (Valiant)**.** *Computing perm(M) for 0-1 matrices M is #P-complete.*

Thus simply removing the apparent complicating factor of $\mathrm{sgn}(\sigma)$ results in a dramatic *increase* in complexity! When M is a 0-1 matrix, $\mathrm{perm}(M)$ has the following combinatorial interpretation. Consider a bipartite graph G with n vertices on each side of the bipartite split, where there is an edge between the i'th vertex on the left-hand side and the j'th vertex on the right-hand side if and only if $M_{ij} = 1$. Then each term $\prod_{i=1}^{n} M_{i,\sigma(i)}$ is equal to 1 if and only if σ corresponds to a perfect matching in G – i.e. a set of n edges such that each vertex is included in exactly one edge. Hence computing the permanent of a 0-1 matrix is equivalent to counting perfect matchings in a bipartite graph, which implies that the problem is in #P. As we saw near the start of the course, the decision variant of this problem (i.e. the problem of determining whether a bipartite graph G has a perfect matching) is in P, so generalising from decision to counting makes the problem considerably more difficult. More generally, we can associate any matrix M with a bipartite undirected graph G, by putting (possibly negative) weights on each edge of G; $\mathrm{perm}(M)$ is then a sum over perfect matchings in G, where each matching is weighted by the product of its edge weights. Valiant's result shows that counting perfect matchings in bipartite graphs is #P-hard. Remarkably, it turns out to be possible to count perfect matchings in arbitrary graphs in polynomial time – as long as the graph is planar (i.e. can be drawn in the plane with no edges crossing). The result that this is possible is known as the FKT algorithm[1] and has significant applications in statistical physics.

Another interpretation of the permanent, which will be important later, is in terms of *cycle covers*. A cycle cover of G is a set of cycles in G such that each vertex is contained in exactly one cycle. It is easy to convince oneself that, if G is a directed graph with adjacency matrix A, $\mathrm{perm}(A)$ is precisely the number of cycle covers of G. Similarly to the bipartite graph interpretation, for any square matrix M, $\mathrm{perm}(M)$ is equal to a sum over weighted cycle covers of a graph corresponding to M. Figure 8.1 illustrates these ideas.

The proof given here that the permanent is #P-complete is based on Papadimitriou's. In order to prove Theorem 82, we first prove that computing the permanent of general integer matrices M is #P-hard, and then reduce to the case where M is a 0-1 matrix.

Theorem 83. *Computing perm(M) for integer matrices M is #P-hard.*

[1] See `http://www.damtp.cam.ac.uk/user/am994/presentations/matchings.pdf`, for example, for a discussion of this beautiful algorithm.

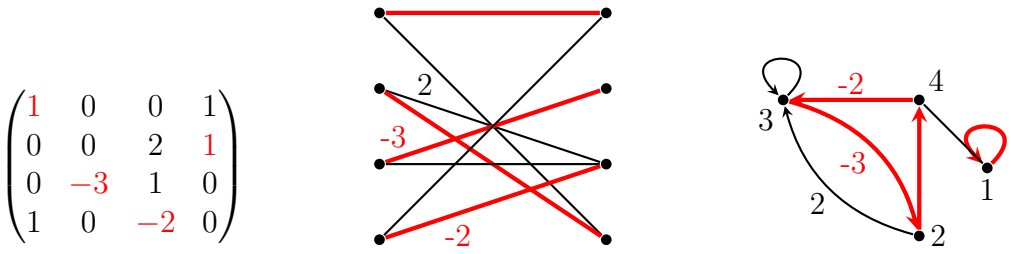


Figure 8.1: A matrix and its correspondence to weighted bipartite and directed graphs. A perfect matching and its corresponding cycle cover are marked in thick red lines.

Figure 8.2: Variable and clause gadgets. Undirected edges (i.e. without arrows) correspond to an arc in each direction. In each cycle cover, exactly one of the two rightwards-moving arcs of each variable gadget must be included, corresponding to true or false. Also, at least one of the external edges of each clause gadget must be *excluded*, corresponding to a variable satisfying that clause.

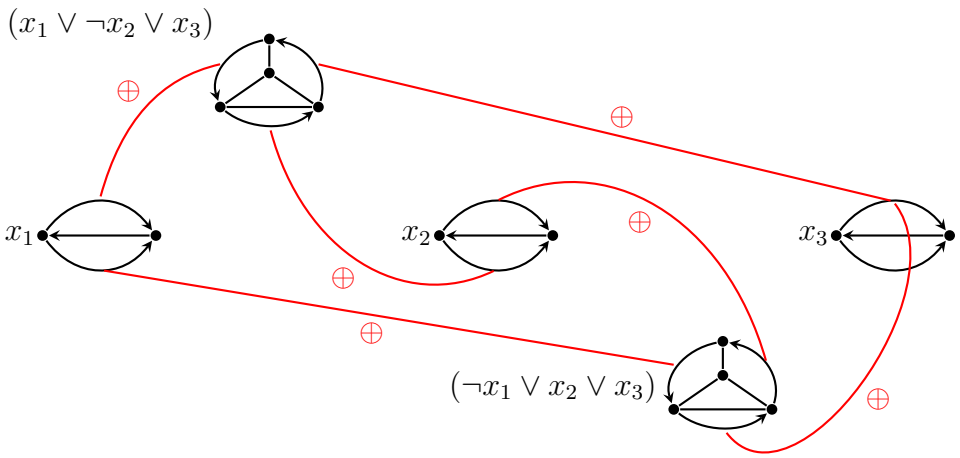
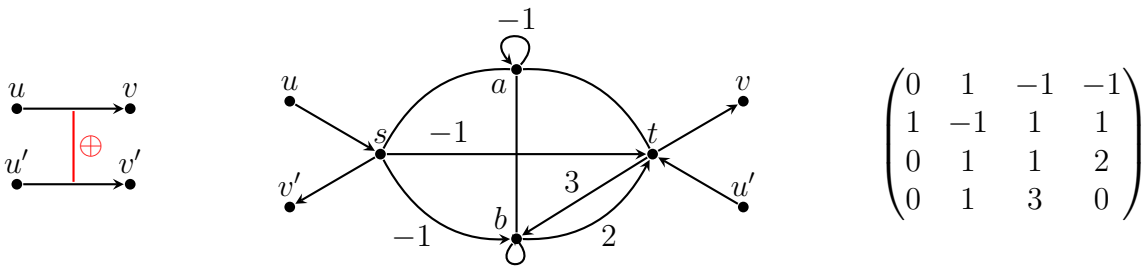


Figure 8.3: Constructing a graph whose cycle covers correspond to satisfying assignments to the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.



$$\begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix}$$

Figure 8.4: Constructing an XOR gate. The graph G_{XOR} is intended to ensure that there is either an edge from u to v, or from u' to v', but not both. Here we describe G_{XOR} by its graph and its adjacency matrix, where vertices are ordered (s, a, b, t).

Proof. We will reduce #3-SAT to computing the permanent. Given a 3-CNF boolean formula ϕ containing m literals, we will produce a matrix M such that $\text{perm}(M) = 4^m \#\phi$, where $\#\phi$ is the number of satisfying assignments of ϕ. In other words, we will construct a weighted graph G such that the sum over weighted cycle covers of G is equal to $4^m \#\phi$.

The construction will use three kinds of gadget: variable, clause and XOR gadgets. Each gadget corresponds to a subgraph as illustrated in Figures 8.2 and 8.4. First consider the graph formed by taking one variable gadget G_i per variable x_i, and one clause gadget G_C per clause C. We would like to make each cycle cover of this graph correspond to a satisfying assignment of ϕ. Label each of the three external arcs of a clause gadget with a literal used in that clause. An external arc being included in a particular cycle cover of G_C is intended to correspond to that literal *not* satisfying C. If a clause contains fewer than three variables, the remaining arcs are left unlabelled. Observe that any cycle cover of G_C must include at most two of the external arcs[2]. Further, it is easy to convince oneself that every choice of a proper subset of the external arcs of G_C (including the empty set) corresponds to exactly one cycle cover of G_C.

Imagine we had a (magical!) "XOR gadget" which we could apply to any pair of arcs, and which would enforce the constraint that exactly one of the two arcs is included in any cycle cover. Given such a gadget, we could calculate $\#\phi$ as follows. Whenever a variable x_i appears in a clause C, apply the XOR gadget between the external arc of G_C corresponding to x_i, and either the "false" arc in G_i (if x_i appears negated in C), or the "true" arc in G_i (if x_i appears un-negated in C). Then each assignment to variables $x_1, \ldots, x_n$ would correspond to exactly one cycle cover (if x satisfies ϕ), or no cycle covers (if x does not satisfy ϕ), so the permanent of the whole graph would equal $\#\phi$. Figure 8.3 illustrates this construction.

We will simulate such an XOR gadget using the construction G_{XOR} illustrated in Figure 8.4, and analyse the permanent of the resulting graph. The permanent is is a sum over weighted cycle covers, and can hence be split up in terms of sums over cycle covers that use different subsets of the four arcs $u \to s$, $s \to v'$, $t \to v$, $u' \to t$.

[2]One may be uneasy that G_C and G_i contain multiple arcs between certain nodes. These extra arcs will disappear shortly; in any case, they can be subsumed into one weighted arc.

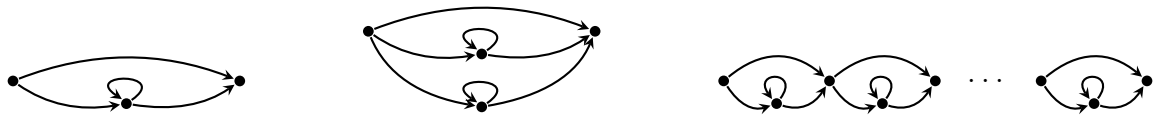


Figure 8.5: Simulating positive weights 2, 3 and 2^n respectively.

First, the sum over cycle covers that use *none* of these four arcs is equal to 0. This is because the permanent of G_{XOR}'s weighted adjacency matrix is equal to 0 (as can be checked), and the permanent of a graph made of two disjoint subgraphs is equal to the product of their respective permanents. Similarly, the sum over cycle covers that use either of the pairs of arcs $(u \to s,\ s \to v')$ and $(u' \to t,\ t \to v)$, or both of these pairs, is also equal to 0. Thus the only cycle covers which contribute to the permanent of the whole graph are those which either contain an arc $u \to s$ and an arc $t \to v$, or contain an arc $u' \to t$ and an arc $s \to v'$, but not both of these pairs. One can calculate that in either of these cases, the weighted sum of cycle covers gets a multiplicative contribution of 4 from G_{XOR}.

This essentially completes the proof: each cycle cover which does not satisfy all of the XOR constraints gives a contribution of 0 to the overall sum, and each cycle cover which does satisfy all the constraints gives a contribution of 4^m (there are m XOR gadgets in total, each contributing a factor of 4). Therefore, the permanent of the adjacency matrix of G is equal to $4^m \#\phi$. □

The above proof appeared to crucially rely on some clever cancellations between positive and negative entries in M. Perhaps surprisingly, we now show that the permanent remains hard even when restricted to 0-1 matrices.

Proof of Theorem 82. First, we observe that any positive integer weights in the graph G corresponding to M can be replaced with 1's using the construction illustrated in Figure 8.5; even exponentially large weights can be dealt with by attaching subgraphs in series. However, there are also some weights equal to -1. To deal with these, we change the problem under consideration: rather than computing the permanent, we instead consider computing the permanent modulo N, for some integer N. This problem is clearly no easier than computing the permanent, so if we can show that #3-SAT reduces to computing the permanent modulo N, we have shown that computing the permanent itself is #P-hard. When working modulo N, $-1 \equiv N - 1$. Take N to be a large integer, $N = 2^n + 1$, where we pick n such that the permanent of M does not exceed N ($n = 8m$ works, for example). Now the -1's in M can be replaced with 2^n's, which can be simulated as before. The number of satisfying assignments of the original formula is then precisely equal to the permanent of the resulting matrix, modulo N. □

8.2 Decision trees

We have seen that there are many conjectured separations between complexity classes which are not known to hold. We now change tack and consider a much simpler model than the Turing machine, where it is much easier to find provable lower bounds on the complexity of problems. Nevertheless, even in this simple model there are still many open questions. We approach this issue via the topic of decision tree complexity. An excellent survey on decision tree complexity and many other measures of complexity for boolean functions is "Complexity measures and decision tree complexity" by Buhrman and de Wolf, which can easily be found online. The results discussed here concerning game trees and graph properties are due to Saks and Wigderson and Rivest and Vuillemin, respectively; the exposition here is largely based on lecture notes for the course "Concrete Models of Computation" by Jeff Erickson[3].

Imagine we want to compute some (known) boolean function $f : \{0,1\}^n \to \{0,1\}$ of some (unknown) input $x \in \{0,1\}^n$. Although x is unknown, we are given access to individual bits of x via an oracle O_x which, on input of an index $i \in \{1,\dots,n\}$, replies with the bit x_i. Our goal is to compute $f(x)$ using the minimal number of queries to O_x in the worst case, i.e. to minimise the maximum (over x) possible number of queries used. It is clear that any function $f(x)$ can be computed using n queries, as this is enough to learn x completely. However, certain functions can be computed with far fewer queries.

For example, consider the function ADDR : $\{0,1\}^{m+2^m} \to \{0,1\}$ where we divide the input of size $n = m+2^m$ into blocks of m bits and 2^m bits (an "address register" and a "data register"), and define $\mathrm{ADDR}(i,x) = x_i$. That is, ADDR returns the bit of the data register given by the address register. It is easy to see that, if we query every bit of the address register, this suffices to compute ADDR on every possible input and uses only $O(\log n)$ queries.

In general, the operation of a deterministic algorithm in the query model is given by a *decision tree* (see Figure 8.6 for a simple example). At each step of the algorithm, we query a bit x_i, where the choice of i depends on the responses to all previous queries we have made. At the end of the algorithm, we either output 0 or 1. Thus we can express the operation of the algorithm by a binary tree whose vertices are labelled with variables x_i, $1 \le i \le n$, and whose leaves are labelled with 0 or 1. The worst-case number of queries made by the algorithm is the depth of the tree, i.e. the maximal number of vertices labelled by variables encountered on any path from the root to a leaf. The minimal depth over all decision trees computing f is called the *decision tree complexity* of f and denoted $D(f)$. More concisely,

$$D(f) = \min_{T, T \text{ computes } f} \max_{x \in \{0,1\}^n} [\text{number of variables queried by } T \text{ on input } x].$$

A function $f : \{0,1\}^n \to \{0,1\}$ is called *evasive* if $D(f) = n$.

We have the following observations about decision tree complexity.

Lemma 84. *For any $f : \{0,1\}^n \to \{0,1\}$ that depends on all n variables, $D(f) = \Omega(\log n)$.*

[3] `http://compgeom.cs.uiuc.edu/~jeffe/teaching/497/05-evasive.pdf`

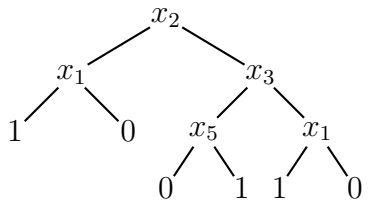


Figure 8.6: A decision tree of depth 3.

Proof. Any decision tree T with depth at most d contains at most $2^{d+1}-1$ internal vertices, so the function computed by T can depend on at most $2^{d+1}-1$ variables. □

Recall that the Hamming weight of a bit-string $y \in \{0,1\}^n$, which we write $\mathrm{wt}(y)$, is the number of 1's in y, i.e. $\mathrm{wt}(y) = |\{i : y_i = 1\}|$.

Lemma 85. *If $|\{x : f(x) = 1\}|$ is odd, then f is evasive. Further, if f is not evasive, then exactly half the bit-strings x such that $f(x) = 1$ have even Hamming weight.*

Proof. For the first part, we prove the contrapositive. For any leaf ℓ at depth d in a decision tree T computing a function $f : \{0,1\}^n \to \{0,1\}$, exactly 2^{n-d} possible input strings x lead to ℓ. If $d < n$, an even number of inputs x lead to ℓ. As each string leads to exactly one leaf, if the depth of T is less than n, $|\{x : f(x) = 1\}|$ is even. For the second part, note that for each such leaf ℓ, exactly half of the strings that lead to ℓ have even Hamming weight. □

One method of lower bounding decision tree complexity is the adversary approach. Here we imagine that the input x is chosen bit by bit, by an adversary, in order to frustrate any possible decision tree making a small number of queries. For example, consider the OR_n function $\mathrm{OR}_n(x_1, \ldots, x_n) = x_1 \vee x_2 \cdots \vee x_n$; we will show that $D(\mathrm{OR}_n) = n$. Let T be a decision tree computing OR_n, and assume (without loss of generality) that T never queries the same variable twice. Whichever variable is queried by T, the adversary will always respond with 0. Even after $n-1$ queries, the algorithm therefore does not know if the answer should be 0 or 1.

The following technique, which is known as the method of polynomials, provides a powerful alternative tool to lower bound decision tree complexity. Any function $f : \{0,1\}^n \to \mathbb{R}$ can be written as a multilinear polynomial $p : \mathbb{R}^n \to \mathbb{R}$ in n variables $x_1, \ldots, x_n \in \mathbb{R}$, and this polynomial is unique. Let $\deg(f)$ be the degree of this polynomial (i.e. the largest total number of variables in any term of p). For example:

- The function computed by the decision tree illustrated in Figure 8.6 has polynomial representation $1 - x_1 - x_2 + x_1x_2 + x_2x_3 + x_2x_5 - x_1x_2x_3 - x_2x_3x_5$ and hence degree 3.
- The function AND_n defined by $\mathrm{AND}_n(x_1, \ldots, x_n) = x_1 \wedge \cdots \wedge x_n$ has polynomial representation $x_1x_2 \ldots x_n$ and hence $\deg(\mathrm{AND}_n) = n$.

Theorem 86. *For any boolean function f, $D(f) \geq \deg(f)$.*

Proof. We show by induction that any decision tree T of depth d that computes a boolean function f gives rise to a degree d polynomial that represents f. This is clearly true for decision trees of depth 0 (which are just constant functions). So assume without loss of generality that x_1 is the first variable queried by T. Then $f(x)$ can be written as $f(x) = (1-x_1)f_0(x)+x_1f_1(x)$ for some functions f_0, f_1 computed by decision trees of depth at most $d-1$. As the degree of f_0 and f_1 is at most $d-1$, the degree of f is at most d. □

8.2.1 Nondeterministic decision tree complexity

Just as in the model of time complexity, there is a nondeterministic analogue of the deterministic decision tree model. It is most natural to express the idea of nondeterminism using the concept of certificates, which are defined as follows.

Let $f : \{0,1\}^n \to \{0,1\}$ be a boolean function, and let $C : S \to \{0,1\}$ be an assignment to some subset $S \subseteq [n]$ of the input variables. We say that C is a 0-certificate (resp. 1-certificate) if $f(x) = 0$ (resp. $f(x) = 1$) whenever the variables in S are assigned according to C. We further say that C is a 0-certificate for f on x, for some $x \in \{0,1\}^n$ such that $f(x) = 0$, if C is a 0-certificate and C is consistent with x (i.e. $C(i) = 0 \Leftrightarrow x_i = 0$). 1-certificates for f on x are defined similarly for x such that $f(x) = 1$.

Let $C_x(f)$ be the size of a smallest $f(x)$-certificate for f on x. Then the 0-certificate complexity of f is $C^{(0)}(f) = \max_{x,f(x)=0} C_x(f)$, and similarly the 1-certificate complexity of f is $C^{(1)}(f) = \max_{x,f(x)=1} C_x(f)$. The certificate complexity of f is $C(f) = \max\{C^{(0)}(f), C^{(1)}(f)\}$.

The idea behind certificates is that if the algorithm knows the bits x_i, $i \in S$, it can be sure that $f(x) = 0$ or $f(x) = 1$, respectively. For example, consider the OR function on n bits. For any x such that $\mathrm{OR}(x) = 1$, knowing that any bit $x_i = 1$ suffices to certify that $\mathrm{OR}(x) = 1$, so $C^{(1)}(\mathrm{OR}) = 1$. However, if $x = 0^n$, in order to certify that $f(x) = 0$ the algorithm needs to know all n bits of the input. Hence $C^{(0)}(\mathrm{OR}) = n$, so $C(\mathrm{OR}) = n$.

Roughly speaking, the class of functions f such that $D(f) = \mathrm{polylog}(n)$ is the decision tree analogue of the time complexity class P; similarly, functions f with $C^{(1)}(f) = \mathrm{polylog}(n)$, $C^{(0)}(f) = \mathrm{polylog}(n)$ correspond to NP and $\mathsf{co\text{-}NP}$ respectively. Recall that, in the world of time complexity, it is unknown whether $\mathsf{P} = \mathsf{NP} \cap \mathsf{co\text{-}NP}$. In the simplified world of decision tree complexity, we can actually prove a result analogous to $\mathsf{P} = \mathsf{NP} \cap \mathsf{co\text{-}NP}$.

Theorem 87. *For any $f : \{0,1\}^n \to \{0,1\}$, $D(f) \leq C^{(0)}(f)C^{(1)}(f) \leq C(f)^2$.*

In order to prove Theorem 87, we will need the following lemma.

Lemma 88. *Let $C_0 : S \to \{0,1\}$, $C_1 : T \to \{0,1\}$ be, respectively, a 0-certificate and a 1-certificate for a boolean function $f : \{0,1\}^n \to \{0,1\}$. Then there exists $i \in S \cap T$ such that $C_0(i) \neq C_1(i)$.*

Proof. Assume the contrary: for all $i \in S \cap T$, $C_0(i) = C_1(i)$. Then consider the bit-string x which assigns $x_i = C_0(i)$ for all $i \in S$, $x_i = C_1(i)$ for all $i \in T$, and $x_i = 0$ for $i \notin S \cup T$. This x is consistent with both C_0 and C_1, contradicting the definition of certificates. □

Proof of Theorem 87. Let $c = C^{(0)}(f)C^{(1)}(f)$. The proof is by induction on c. For $c = 1$, by Lemma 88 f depends only on one input bit x_i, so we must have $f(x) = x_i$ or $f(x) = \neg x_i$. In either case $D(f) = 1$. So assume that $C^{(0)}(f) = r \geq 2$ and let $C : S \to \{0,1\}$ be a minimal 0-certificate, for some S of size r. Our decision tree for f on input x will begin by querying all the variables in S. If the answers were consistent with C, then $f(x) = 0$, so we can stop. Otherwise, we recursively compute the function $g(x')$ formed by fixing the bits of x consistent with the responses to the queries. By Lemma 88, S intersects every T such that $C' : T \to \{0,1\}$ is a 1-certificate of g. This implies that we can save one query when querying any 1-certificate C' of g (as we already know one bit of C'). Hence, as $C^{(0)}(g) \leq C^{(0)}(f)$ and $C^{(1)}(g) \leq C^{(1)}(f)$, by the inductive hypothesis

$$D(f) \leq r + C^{(0)}(g)(C^{(1)}(g) - 1) \leq r + r(C^{(1)}(f) - 1) \leq C^{(0)}(f)C^{(1)}(f),$$

which is the theorem. □

8.2.2 Randomised decision tree complexity

Also by analogy with the model of time complexity, we can generalise the decision tree model by adding the ability to use randomness. A probabilistic decision tree can be defined as follows. At each step, rather than choosing a variable to query deterministically based on previous queries, the algorithm can choose a variable to query based on a string of random bits, as well as the answers to previous queries. An alternative way of arriving at the same definition is to define a probabilistic decision tree as a probability distribution over deterministic decision trees. The depth of a probabilistic tree T is defined as the worst-case expected number of variables queried. That is, the maximum over all x of the expected number of variables queried by T on input x.

For $f : \{0,1\}^n \to \{0,1\}$, we then let $R(f)$ denote the randomised decision tree complexity of f – i.e. the minimal depth of T, over all randomised decision trees T that compute f. As with deterministic decision trees, we insist that T computes f with certainty. This restriction may appear to imply that no saving in queries can be made over deterministic trees, as (unlike the complexity class BPP, for instance) probabilistic trees must still always be correct on every input. However, this is not the case: we now give an example for which probabilistic trees significantly outperform deterministic trees.

Let $\mathrm{NAND}_n : \{0,1\}^{2^n} \to \{0,1\}$ denote the n-level NAND tree function, which is the function defined recursively by $\mathrm{NAND}_1(x_1, x_2) = \neg(x_1 \wedge x_2)$, and

$$\mathrm{NAND}_n(x_1, \ldots, x_{2^n}) = \neg(\mathrm{NAND}_{n-1}(x_1, \ldots, x_{2^{n-1}}) \wedge \mathrm{NAND}_{n-1}(x_{2^{n-1}+1}, \ldots, x_{2^n})).$$

Theorem 89. *$D(NAND_n) = 2^n$, but $R(NAND_n) = O\left(\left(\frac{1+\sqrt{33}}{4}\right)^n\right)$.*
Hence $R(NAND_n) = O(D(NAND_n)^{0.753\ldots})$.

Proof. The claim that $D(\mathrm{NAND}_n) = 2^n$ follows from $\deg(\mathrm{NAND}_n) = 2^n$, which can be proven by induction. For the second claim, consider the following randomised recursive algorithm $\mathcal{A}_n$ for computing NAND_n. Beginning at the root, choose one of the two children

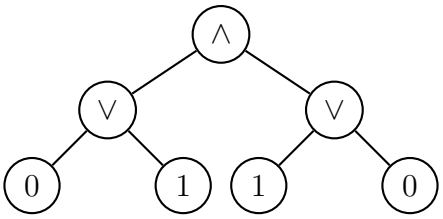


Figure 8.7: An AND-OR tree corresponding to a game with a winning strategy for player 1.

at random and evaluate its subtree using $\mathcal{A}_{n-1}$. If it evaluates to 0, then return 1. Otherwise, evaluate the other subtree using $\mathcal{A}_{n-1}$.

We now analyse the expected number of queries made by this algorithm. Let $\alpha_0(n)$ (resp. $\alpha_1(n)$) denote the maximum expected number of queries over all inputs that evaluate to 0 (resp. 1). It is obvious that

$$\alpha_0(n) \leq 2\alpha_1(n-1)$$

(for inputs which evaluate to 0, both the subtrees must evaluate to 1). On the other hand, for any x, if $\mathrm{NAND}_n(x) = 1$, then with probability at least $1/2$ we evaluate only one of the subtrees. So

$$\alpha_1(n) \leq \alpha_0(n-1) + \frac{1}{2}\alpha_1(n-1).$$

So $\alpha_1(n) \leq 2\alpha_1(n-2) + \frac{1}{2}\alpha_1(n-1)$. Solving this recurrence gives $\alpha_1(n) \leq \left(\frac{1+\sqrt{33}}{4}\right)^n$; $\alpha_0(n)$ then obeys the same bound, up to a constant factor. □

The NAND tree example may appear somewhat contrived. However, it is essentially equivalent to a natural model for the analysis of two-player games. Consider a game like chess where the players take it in turns to make a move. Some sequences of moves lead to victory for player 1, and others to victory for player 2. For simplicity, restrict to the special case where each player only has a choice of two moves (labelled 0 or 1) at each step. In a game with only one round, player 1 wins if either of these moves is a winning move. In a game with two rounds, player 1 wins if, for each move of player 2, there exists a move player 1 can make such that the sequence of two moves leads to victory for player 1. Continuing this process, we find that in a game with n rounds (where n is even), whether player 1 wins can be determined by evaluating an AND-OR tree with n levels (an AND-OR tree is a tree whose root is labelled with AND, and vertices at each level are alternately labelled with OR and AND; see Figure 8.7). The leaves of the tree each correspond to a sequence of moves, and are labelled with bits which specify whether each strategy is a winning strategy for player 1. Finally, observe that AND-OR trees with $2n$ levels are the same as NAND trees with n levels (up to negating the input and output bits). We therefore see that the randomised algorithm of Theorem 89 gives a more efficient way of evaluating game trees.

It is not known how large the separation between randomised and deterministic decision tree complexity can be. However, it is known that the gap can be no bigger than quadratic.

Theorem 90. *For any function $f : \{0,1\}^n \to \{0,1\}$, $D(f) = O(R(f)^2)$.*

Proof. Follows from Theorem 87 by observing that $R(f) \geq C(f)$: if a randomised algorithm is required to succeed with certainty, the bits it queries must form a certificate for f. □

It is also possible to define a variant of randomised decision tree complexity which is perhaps closer in spirit to BPP, where the algorithm is allowed to fail with probability at most ϵ in the worst case, for some $\epsilon < 1/2$; then the model of randomised decision tree complexity discussed above corresponds to $\epsilon = 0$. It is known that, for constant ϵ, there can be at most a cubic separation between this model of randomised decision tree complexity and deterministic decision tree complexity.

8.2.3 Decision trees and symmetry

We now study the question of putting lower bounds on decision tree complexity in the case where f satisfies some symmetry properties. Let $\pi \in S_n$ be a permutation of the integers $\{1, \ldots, n\}$ and for $x \in \{0,1\}^n$ write $\pi(x) = (x_{\pi(1)}, \ldots, x_{\pi(n)})$. We say that $f : \{0,1\}^n \to \{0,1\}$ is invariant under π if $f(x) = f(\pi(x))$ for all x. The set of permutations under which f is invariant of course forms a subgroup of S_n. We say that a group of permutations Γ is transitive if, for any pair $i, j \in [n]$, there exists $\pi \in \Gamma$ such that $\pi(i) = j$.

Intuitively, functions which are invariant under a transitive permutation group "look the same" in every direction, so they should require many queries to compute. This is formalised as the following conjecture, which has been open for several decades.

Conjecture 91. *Every non-constant monotone function $f : \{0,1\}^n \to \{0,1\}$ such that f is invariant under a transitive permutation group is evasive.*

A boolean function f is said to be monotone if flipping the value of an input bit from 0 to 1 cannot change the output of the function from 1 to 0. The restriction to monotone functions in Conjecture 91 is necessary (below we give an example of a non-monotone function which is invariant under a transitive permutation group, but is not evasive).

Conjecture 91 (in fact, a stronger result than this conjecture) is known to be true given certain restrictions on the input size.

Theorem 92. *Let n be a prime power. If $f : \{0,1\}^n \to \{0,1\}$ is invariant under a transitive permutation group Γ and $f(0^n) \neq f(1^n)$, f is evasive.*

Note that, if f is monotone and non-constant, $f(0^n) \neq f(1^n)$.

Proof. Let $n = p^r$ for some prime p. Write orbit(x) for the orbit of x under Γ, i.e. $\{y : y = \pi(x), \pi \in \Gamma\}$. We first show that, for all $x \notin \{0^n, 1^n\}$, |orbit(x)| is an integer multiple of p.

We have

$$\sum_{y \in \text{orbit}(x)} \text{wt}(y) = \sum_{y \in \text{orbit}(x)} \sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \sum_{y \in \text{orbit}(x)} y_i = n \sum_{y \in \text{orbit}(x)} y_1,$$

where the last equality holds because Γ is transitive, so there must exist $\pi \in \Gamma$ such that $\pi(i) = 1$. On the other hand, all bit-strings $y \in \text{orbit}(x)$ have the same Hamming weight, so

$$\sum_{y \in \text{orbit}(x)} \text{wt}(y) = |\text{orbit}(x)|\text{wt}(x).$$

Thus $|\text{orbit}(x)|\text{wt}(x)$ is a multiple of $n = p^r$. As $\text{wt}(x)$ is not a multiple of n for $x \notin \{0^n, 1^n\}$, $|\text{orbit}(x)|$ is a multiple of p.

By Lemma 85, f is evasive if

$$\sum_{x, f(x)=1} (-1)^{\text{wt}(x)} \neq 0.$$

For each x such that $f(x) = 1$, consider the contribution of $\text{orbit}(x)$ to this sum. All bit-strings $y \in \text{orbit}(x)$ have $f(y) = 1$, $\text{wt}(y) = \text{wt}(x)$, so

$$\sum_{y \in \text{orbit}(x)} (-1)^{\text{wt}(x)} = |\text{orbit}(x)|(-1)^{\text{wt}(x)}.$$

For $x \notin \{0^n, 1^n\}$, this quantity is a multiple of p. Since either $f(0^n) = 1$ or $f(1^n) = 1$, but not both, the sum over all x is a multiple of p, plus or minus 1. In particular, the sum is nonzero, so f is evasive. □

A particularly interesting set of boolean functions to consider in the decision tree model are graph properties. A graph property is a function $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$, where the input bits represent the edges in an undirected graph, such that f is unchanged under permutations of the vertices of the graph. Natural graph properties include connectivity (is every vertex connected to every other vertex by an edge?) and whether a graph has a Hamiltonian path. Most such properties which have been studied are known to be evasive. However, it is known that non-evasive graph properties do exist.

A *scorpion graph* on n vertices contains a vertex b (the body) with $n-2$ neighbours, a vertex s (the sting) with 1 neighbour, and a vertex t (the tail) connected to both b and s (and only them). The remaining vertices form a set S (the feet) which are all connected to b. Each pair of vertices within S may or may not have an edge between them. See Figure 8.8 for an illustration.

Theorem 93. *There is a deterministic algorithm which determines whether a graph G is a scorpion graph using $O(n)$ queries.*

Proof. Exercise. □

We therefore consider only *monotone* graph properties and would like to show that any monotone graph property is evasive (i.e. the special case of Conjecture 91 where f is a graph property). Theorem 92 does not appear to immediately apply to graph properties, as $\binom{n}{2}$ is never a power of a prime for $n > 3$. However, we can use the theorem to prove the following result.

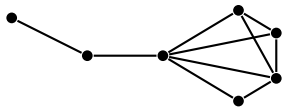


Figure 8.8: A scorpion graph.

Theorem 94. *Assume $n = 2^k$ for some integer k. Then every non-trivial monotone graph property $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ satisfies $D(f) \geq n^2/4$.*

Proof. For any integer j, let G_j be the graph made up of $n/2^j$ disjoint copies of a clique on 2^j vertices. Thus G_0 is the empty graph and G_k is the complete graph on n vertices. As f is monotone and non-trivial, $f(G_0) = 0$ and $f(G_k) = 1$; further, there is a unique i such that $f(G_i) = 0$ but $f(G_{i+1}) = 1$.

Imagine we are given the promise that the input graph G has the following property: the induced subgraph on vertices $1, \ldots, n/2$ consists of $n/2^{i+1}$ disjoint cliques, as does the induced subgraph on vertices $n/2 + 1, \ldots, n$. This implies that the only input bits (i.e. edges) on which f depends are the $n^2/4$ potential edges between the first $n/2$ vertices and the second $n/2$ vertices. Let $g : \{0,1\}^{n^2/4} \to \{0,1\}$ denote the restriction of f to these edges. Observe that g is still monotone. Since n is a power of 2, $n^2/4$ is a power of a prime. $g(0^{n^2/4}) \neq g(1^{n^2/4})$ since $g(0^{n^2/4}) = f(G_i) = 0$ and $g(1^{n^2/4}) = 1$ (as this corresponds to G_{i+1} with some additional edges). Also, g is invariant under the transitive permutation group generated by swapping any two vertices either side of the bipartite cut. Thus g is evasive by Theorem 92, so $D(f) \geq D(g) = n^2/4$. □

Using similar but more technical ideas, the following result can be proven.

Theorem 95. *Let $f : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ be a non-trivial monotone graph property. Then $D(f) = \Omega(n^2)$.*

However, even for monotone graph properties, Conjecture 91 remains open.

8.3 Approximation algorithms and probabilistically checkable proofs

Thus far many of the problems we have considered have been decision problems, i.e. problems with a yes-no answer. Many problems we wish to solve in practice have a different flavour and can be understood as *optimisation* problems. For example:

- KNAPSACK: given a set of n *values* $\{v_i\}$ and *weights* $\{w_i\}$, and a maximum weight M, maximise $\sum_{i=1}^n v_i x_i$, subject to $\sum_{i=1}^n w_i x_i \leq M$, $x_i \in \{0,1\}$.
- TRAVELLING SALESMAN: given a graph G whose edges are weighted with costs, find a tour (cycle) that visits all vertices exactly once and achieves the lowest cost.

- MAX-3SAT: given a CNF boolean formula (not necessarily satisfiable) where each clause contains at most 3 variables, find an assignment to the variables that satisfies the largest number of clauses.
- MAX-E3SAT: the same as MAX-3SAT, but where each clause contains *exactly* 3 variables.

The decision variants of all of these are NP-complete, so we do not expect to be able to solve them exactly in polynomial time. But often in practice we might be satisfied with a "reasonably" approximate answer. The concept of the quality of an approximation can be formalised as follows.

Every optimisation problem P has a set of *feasible solutions* $F(x)$ for each input x, and each solution $s \in F(x)$ has a *value* $v(s)$[4]. If P is a *maximisation* problem, the optimum value is then defined as

$$\mathrm{OPT}(x) = \max_{s \in F(x)} v(s);$$

and if P is a *minimisation* problem,

$$\mathrm{OPT}(x) = \min_{s \in F(x)} v(s).$$

Let $\mathcal{A}$ be an algorithm which, given an instance x, returns a feasible solution $\mathcal{A}(x) \in F(x)$. If P is a maximisation problem, we say that $\mathcal{A}$ is an α-approximation algorithm if, for all x, we have

$$\mathcal{A}(x) \geq \alpha \, \mathrm{OPT}(x),$$

and if P is a minimisation problem, we say that $\mathcal{A}$ is an α-approximation algorithm if, for all x, we have

$$\mathcal{A}(x) \leq \alpha \, \mathrm{OPT}(x).$$

For a maximisation (resp. minimisation) problem P, the *approximation threshold* of P is the largest (resp. smallest) α such that there exists a polynomial-time α-approximation algorithm for P. Beware that there is no standardised terminology in this field; for example, some authors redefine $\alpha \mapsto 1/\alpha$ for maximisation problems, so it is always at least 1.

Intuitively, the approximation threshold of a problem is how well we can expect to solve it in practice. It will turn out that different NP-complete problems can have very different approximation thresholds.

8.3.1 The good: arbitrarily close approximation

Theorem 96. *There is a polynomial-time* $(1 - \epsilon)$*-approximation algorithm for* KNAPSACK*, for any* $\epsilon > 0$.

[4]This terminology makes sense if we want to maximise the value; for minimisation problems the term "cost" might be more appropriate.

Proof. We first use dynamic programming to get a pretty good algorithm for solving KNAPSACK. Let $V = \max\{v_1, \ldots, v_n\}$ be the maximum value of an item. Define a $(n+1)\times(nV+1)$ table W by letting $W(i, v)$ be the minimum weight attainable by selecting some subset of the first i items, such that their total value is exactly v. $W(0, v) = \infty$ for all v, and then

$$W(i+1, v) = \min\{W(i, v), W(i, v - v_{i+1}) + w_{i+1}\}.$$

Once we have filled in the table, we pick the largest v such that $W(n, v) \leq M$. This algorithm solves KNAPSACK in time $O(n^2V)$. Note that this is *not* polynomial in the input size, as V might be very large.

We can make this algorithm run faster, at the cost of introducing inaccuracy, by attempting to reduce V. To do this, we simply truncate the last b bits of each value v_i (for some b to be determined) and replace them with zeroes, which can then be ignored. That is, we get new values $v'_i = 2^b\lfloor v_i/2^b \rfloor$. We end up with an algorithm that runs in $O(n^2V/2^b)$ time. How bad is the solution we get? Defining S and S' to be the original and new solutions (i.e. sets of items), we have (exercise!)

$$\sum_{i\in S} v_i \geq \sum_{i\in S'} v_i \geq \sum_{i\in S'} v'_i \geq \sum_{i\in S} v'_i \geq \sum_{i\in S}(v_i - 2^b) \geq \sum_{i\in S} v_i - n\,2^b.$$

Therefore, as V is a lower bound on the value of the optimal solution, this algorithm runs in time $O(n^2V/2^b)$ and outputs a solution that is at most a $(1-\epsilon)$ fraction of the optimum, where $\epsilon = n2^b/V$.

But this implies that, for any ϵ, if we choose b such that $2^b = V\epsilon/n$, we obtain an algorithm running in $O(n^3/\epsilon)$, which is polynomial in n for any fixed ϵ. □

8.3.2 The bad: arbitrary inapproximability

Theorem 97. *Unless* $\mathsf{P} = \mathsf{NP}$*, there is no polynomial-time* $(1+\epsilon)$*-approximation algorithm for* TRAVELLING SALESMAN*, for any* $\epsilon > 0$*.*

Proof. For a contradiction, suppose there is such an algorithm, and call it $\mathcal{A}$. We will use it to construct a polynomial-time algorithm for the NP-complete HAMILTONIAN CYCLE problem.

Given a graph G with n vertices, the algorithm for HAMILTONIAN CYCLE constructs a TRAVELLING SALESMAN instance with n vertices, where the distance between vertices i and j is 1 if there is an edge in G between i and j, and $(1+\epsilon)n$ otherwise. We now apply our approximation algorithm $\mathcal{A}$ to this problem.

If $\mathcal{A}$ returns a tour of cost exactly n, then we know that G has a Hamiltonian cycle. On the other hand, if $\mathcal{A}$ returns a tour with one or more edges of cost $(1+\epsilon)n$, then the total length of the tour is strictly greater than $(1+\epsilon)n$. As we have assumed that $\mathcal{A}$ never returns a tour with cost greater than $(1+\epsilon)$ times the optimum cost, this means that there is no tour with cost n or less. Therefore, G does not have a Hamiltonian cycle. □

8.3.3 The ugly: approximability up to a constant factor

Theorem 98. *Unless* $\mathsf{P} = \mathsf{NP}$*, there is no polynomial-time* $(\frac{7}{8} + \epsilon)$*-approximation algorithm for* Max-E3Sat*, for any* $\epsilon > 0$*.*

There is an easy randomised $\frac{7}{8}$-approximation algorithm for this problem: just pick an assignment to the variables at random. A random assignment will satisfy each clause with probability $\frac{7}{8}$. By linearity of expectation, this implies that if there are m clauses, on average $\frac{7}{8}m$ clauses will be satisfied. (This algorithm does not work for the more general Max-3Sat problem. There is in fact also a $\frac{7}{8}$-approximation algorithm for Max-3Sat; this algorithm is based on semidefinite programming and its proof of correctness is considerably more complicated.) Theorem 98 says that one cannot do any better than this trivial algorithm, unless $\mathsf{P} = \mathsf{NP}$.

We will prove a weaker variant of this result, which can be stated as follows.

Theorem 99. *Unless* $\mathsf{P} = \mathsf{NP}$*, there is an* $\epsilon > 0$ *such that there is no polynomial-time* $(1 - \epsilon)$*-approximation algorithm for* Max-3Sat*.*

Note that we have lost the tight bound on the approximation ratio and have also generalised from Max-E3Sat to Max-3Sat. The proof of Theorem 99 depends on a recently developed, and apparently unrelated, concept in theoretical computer science: probabilistically checkable proofs (PCPs).

8.3.4 Probabilistically checkable proofs

Imagine we have a proof system where a prover wishes to convince a verifier that the verifier's input is in a language $\mathcal{L}$. Given an n-bit input x and a poly(n)-bit proof, the verifier first looks at x and performs polynomial-time computation on x. The verifier then chooses, at random, a *constant* number of bits of the proof to read. The verifier reads these bits and performs some test on them, either accepting or rejecting the input depending on the result of the test.

The **PCP Theorem** states that any language $\mathcal{L} \in \mathsf{NP}$ has a proof system of this form where:

- For all $x \in \mathcal{L}$, there exists a proof such that the verifier accepts with probability 1.
- For all $x \notin \mathcal{L}$, for any proof, the verifier accepts with probability at most $1/2$.

In comparison with the usual characterisation of NP, we are looking at much fewer bits of the proof, but at the expense of having some probability of incorrectly identifying strings as being in $\mathcal{L}$ that are actually not in $\mathcal{L}$. One can generalise the notion of PCPs by writing

$$\mathcal{L} \in \mathsf{PCP}_{c,s}(r(n), q(n))$$

to mean that there exists a probabilistic polynomial-time algorithm V which has access to $r(n)$ random bits, and may make $q(n)$ queries to a proof oracle π, such that

- **(Completeness)** For all $x \in \mathcal{L}$, there exists a proof π such that the probability that V outputs 1 on π is at least $c(n)$.
- **(Soundness)** For all $x \notin \mathcal{L}$ and for all proofs π, the probability that V outputs 1 on π is at most $s(n)$.

Then it is immediate from the definition of NP that $\mathsf{NP} = \mathsf{PCP}_{1,0}(0, \text{poly}(n))$, while the PCP Theorem is that $\mathsf{NP} \subseteq \mathsf{PCP}_{1,1/2}(O(\log n), O(1))$.

Arora-Barak has much more information about PCPs (Chapters 11 and 22) than we can cover here, including the proof of the tight bound for Max-E3sat; there are even entire lecture courses on the subject[5]. The original proof of the PCP Theorem was the culmination of a series of results drawing together many different concepts in theoretical computer science. Ryan O'Donnell has written a nice history of this[6]. An excellent introduction to the beautiful subject of Fourier analysis of boolean functions is provided by lecture notes from a course on the topic, again by Ryan O'Donnell[7].

8.3.5 The Max-qCSP problem

The proof of the full PCP Theorem is technical, and beyond the scope of this course. However, we will prove two related results. First, that the PCP Theorem implies an inapproximability result for Max-3sat. Second, we will prove a weaker variant of the PCP Theorem where the proof is exponentially long (but the verifier still only looks at a constant number of bits). To prove the first part, we first introduce a more general problem: Max-qCSP. A qCSP ("q-ary constraint satisfaction problem") instance is a collection of boolean functions $\phi_i : \{0,1\}^n \to \{0,1\}$ called *constraints*, each of which depends on at most q of its input bits. The Max-qCSP problem is to find an input x that satisfies the largest possible number of these constraints. Max-3sat is the special case of Max-3CSP where the constraints are only allowed to be OR functions (perhaps with negations of the input bits).

We will show that the PCP Theorem implies inapproximability of Max-qCSP, for some $q = O(1)$. Consider a language $\mathcal{L} \in \mathsf{NP}$. Given an n-bit input x, using the proof whose existence is guaranteed by the PCP Theorem, we will construct an instance ϕ of Max-qCSP with $m = \text{poly}(n)$ constraints, such that:

- If $x \in \mathcal{L}$, ϕ is satisfiable;
- If $x \notin \mathcal{L}$, the largest number of constraints of ϕ that can be satisfied is at most $\frac{m}{2}$.

Consider a claimed proof that $x \in \mathcal{L}$, and introduce $\text{poly}(n)$ variables $\{y_i\}$, with one variable corresponding to each bit of the proof. Let R be a string of random bits generated by the verifier. Then R corresponds to a set of bits of the proof to read, and a test function f_R to perform on them. For each R, we produce a constraint corresponding to f_R. This constraint

[5] http://www.cs.washington.edu/education/courses/cse533/05au/

[6] http://www.cs.washington.edu/education/courses/533/05au/pcp-history.pdf

[7] http://www.cs.cmu.edu/~odonnell/boolean-analysis/

depends on the variables that are read when random string R is generated, and is satisfied if and only if f_R passes. As the verifier only looks at a constant number of bits of the proof, there are at most $O(\log n)$ random bits used, so at most $\text{poly}(n)$ constraints will be produced.

Now, if $x \in \mathcal{L}$, by the PCP Theorem all the constraints are satisfied. On the other hand, if $x \notin \mathcal{L}$, at least half of the constraints must not be satisfied (or the probability of the verifier accepting would be greater than $\frac{1}{2}$).

But this reduction implies that, if we could distinguish between satisfiable instances of MAX-qCSP and those for which at most half the constraints are satisfiable, we could solve any problem in NP, and hence P = NP. This in turn means that MAX-qCSP cannot be approximated within a ratio of $\frac{1}{2}$, unless P = NP.

8.3.6 Back to MAX-3SAT

The final step in the proof of Theorem 99 is to show that the result of the previous section implies inapproximability of MAX-3SAT, which we will achieve by converting the MAX-qCSP instance ϕ into a MAX-3SAT instance ϕ'. First, using the universality of CNF formulae (Theorem 11), we can rewrite each of the clauses in ϕ as a q-ary CNF expression with at most 2^q clauses. Second, each of the clauses in this CNF expression can be rewritten as the AND of at most q clauses of length at most 3, using the standard reduction technique from SAT to 3-SAT. We therefore end up with a CNF expression ϕ' with at most $m' = q\,2^q\,m$ clauses, each of which is of length at most 3.

Now it is clear that, if ϕ is satisfiable, ϕ' is also satisfiable. On the other hand, if an ϵ fraction of the constraints of ϕ are not satisfied, then at least an $\frac{\epsilon}{q\,2^q}$ fraction of the clauses in ϕ' are not satisfied (as each constraint in ϕ corresponds to at most $q\,2^q$ clauses in ϕ'). In particular, we see that the ability to find an approximate solution to MAX-3SAT that is at least a $(1 - (q\,2^{q+1})^{-1})$ fraction of the optimum implies the ability to find an approximate solution to MAX-qCSP that is at least a $\frac{1}{2}$ fraction of the optimum. Taking $\epsilon = (q\,2^{q+1})^{-1}$, we have proven Theorem 99: there is a constant $\epsilon > 0$ such that there is no polynomial-time $(1-\epsilon)$-approximation algorithm for MAX-3SAT, unless P = NP.

8.3.7 A weaker version of the PCP theorem

We now turn to proving the promised "toy version" of the PCP theorem. The result we will prove is the following.

Theorem 100. *For any language $\mathcal{L} \in$ NP, there exists a proof system consisting of proofs of exponential length in the input size, where membership in $\mathcal{L}$ can be decided with constant probability by reading only a constant number of bits of the proof. Formally,* $\text{NP} \subseteq \bigcup_{c>0} \text{PCP}_{1,1/2}(n^c, O(1))$.

This is much weaker than the best possible result that can be proven for proof systems with exponentially long proofs; in fact, it is known that $\bigcup_{c>0} \text{PCP}_{1,1/2}(n^c, O(1)) = \text{NEXP}$. The rough idea of the proof of Theorem 100 is as follows.

1. It suffices to give a probabilistically checkable proof of the above form for a single NP-complete problem. We will use a problem called Quadratic Equations over $\mathbb{F}_2$, which is defined as follows. Given a system S of m quadratic equations in n variables $x_1, \ldots, x_n$ over $\mathbb{F}_2$, where the k'th equation is of the form

$$\bigoplus_{i,j=1}^{n} A_{ij}^{(k)} x_i x_j = b_k,$$

   is there an assignment to the variables which satisfies all the equations?

2. Imagine there does exist such a satisfying assignment u, and that the the verifier has access to an oracle $f_u : \{0,1\}^{n^2} \to \{0,1\}$ such that $f_u(A) = \bigoplus_{i,j=1}^{n} A_{ij} u_i u_j$. Then he can check that u is really a satisfying assignment by evaluating f_u on random linear combinations of the $A^{(k)}$ matrices, and verifying that the answer is as expected.

3. The proof will thus consist of evaluations of $f_u(A)$ for all possible $A \in \{0,1\}^{n^2}$. Of course, the verifier cannot trust this proof and must verify that it is of the required form.

4. In order to make it possible for the form of the proof to be checked with only a constant number of queries, it is encoded using an error-correcting code with good error-detection properties.

First, we show that Quadratic Equations really is a good problem to use.

Lemma 101. Quadratic Equations *is* NP*-complete.*

Proof. We show that Circuit Satisfiability $\leq_{\mathsf{P}}$ Quadratic Equations. The proof is similar to the arithmetisation of SAT. Given a circuit C using n input variables $x_1, \ldots, x_n$ and m gates, introduce variables $x_{n+1}, \ldots, x_{n+m}$, one for each of the gates. For each AND gate $x_i = x_j \wedge x_k$, add an equation $x_i = x_j x_k$; for each OR gate $x_i = x_j \vee x_k$, add an equation $x_i = 1 - (1 - x_j)(1 - x_k)$; for each NOT gate $x_i = \neg x_j$, add an equation $x_i = 1 - x_j$. Finally, for the output gate x_{n+m}, set $x_{n+m} = 1$. Then this system of equations has a solution if and only if C is satisfiable. □

We now introduce the code that the prover will use for satisfying assignments, which we call the linear encoding[8]. Each $x \in \{0,1\}^n$ will be encoded as a 2^n-bit codeword given by evaluating the function $L_x : \{0,1\}^n \to \{0,1\}$ defined by

$$L_x(y) = x \cdot y = \bigoplus_{i=1}^{n} x_i y_i$$

on each possible string $y \in \{0,1\}^n$. This is a very inefficient encoding as it leads to an exponential blow-up in the size of x. A function $f : \{0,1\}^n \to \{0,1\}$ is said to be linear if $f(y \oplus z) = f(y) \oplus f(z)$; this terminology justifies the name of the above encoding, as follows.

[8] In the literature this is often called the Hadamard or Walsh-Hadamard code.

Lemma 102. *For all $x \in \{0,1\}^n$, $L_x : \{0,1\}^n \to \{0,1\}$ is linear. Furthermore, every linear function $f : \{0,1\}^n \to \{0,1\}$ is a member of the set $\{L_x : x \in \{0,1\}^n\}$.*

Proof. For the first part, we have

$$L_x(y \oplus z) = \bigoplus_{i=1}^{n} x_i(y_i \oplus z_i) = \left(\bigoplus_{i=1}^{n} x_i y_i\right) \oplus \left(\bigoplus_{i=1}^{n} x_i y_i\right) = L_x(y) \oplus L_x(z)$$

as required. For the second part, consider the set of bit-strings $\{e^i : i \in [n]\}$ where e^i has a 1 at the i'th position, and 0's elsewhere. If $f : \{0,1\}^n \to \{0,1\}$ is linear, then for any $x \in \{0,1\}^n$,

$$f(x) = \bigoplus_{i,x_i=1} f(e^i) = \bigoplus_{i=1}^{n} f(e^i)x_i = L_y(x),$$

where $y \in \{0,1\}^n$ is defined by $y_i = 1 \Leftrightarrow f(e^i) = 1$. □

As $L_x(y) = L_y(x)$, one can also view each function $L_x(y)$ as encoding the values that every possible linear function L_y takes on input x.

Lemma 103. *For all $x, y \in \{0,1\}^n$ such that $x \neq y$, $|\{z : L_x(z) \neq L_y(z)\}| = 2^{n-1}$.*

Proof. The claim follows from observing that the $\{L_x\}$ functions are closely related to the characters $\chi_S(x) = (-1)^{\sum_{i \in S} x_i}$ discussed previously in Section 6.1. Indeed, if $s \in \{0,1\}^n$ corresponds to $S \subseteq [n]$ via $s_i = 1 \Leftrightarrow i \in S$, we have $\chi_S(x) = (-1)^{L_s(x)}$ for all x. As (by Lemma 60) characters corresponding to different subsets are orthogonal, linear functions corresponding to different bit-strings x and y satisfy $|\{z : L_x(z) \neq L_y(z)\}| = 2^{n-1}$. □

Imagine the prover has a claimed solution u. He applies the linear encoding to u, and also to the n^2-bit string v formed by $v_{ij} = u_i u_j$, to give a $(2^n + 2^{n^2})$-bit proof (L_u, L_{uu^T}). (Of course, the verifier cannot be sure that the proof is really of this form, but must check it for himself.) Just as L_u encodes all linear functions of u, L_v encodes all quadratic functions of u; indeed, for $A \in \{0,1\}^{n^2}$,

$$L_v(A) = \bigoplus_{i,j=1}^{n} A_{ij} u_i u_j.$$

This will enable the verifier to check an arbitrary quadratic function of his choice.

It will be convenient to treat these two bit-strings corresponding to all possible evaluations of L_u, L_v as oracles $f : \{0,1\}^n \to \{0,1\}$, $g : \{0,1\}^{n^2} \to \{0,1\}$ which the verifier is allowed to query on arbitrary inputs. The verifier first determines whether f and g really are valid linear encodings of some original bit-strings. By Lemma 102, this is equivalent to determining whether they are linear functions. Therefore, it is natural to use the following *linearity test* to decide this.

Definition 20 (Linearity test)**.** *Given access to a function $f : \{0,1\}^n \to \{0,1\}$, pick $x, y \in \{0,1\}^n$ at random and compute $f(x)$, $f(y)$ and $f(x \oplus y)$. Accept if $f(x) \oplus f(y) = f(x \oplus y)$; otherwise reject.*

As the linearity test only makes a few queries to f and g, it is intuitively clear that it will not be able to reject all nonlinear functions, but only those which are far from linear in some sense. For functions $f, g : \{0,1\}^n \to \{0,1\}$, we say that g is ϵ-far from f if $|\{x : f(x) \neq g(x)\}| = \epsilon 2^n$. For a family of functions $\mathcal{F}$, we say that g is ϵ-far from $\mathcal{F}$ if $\min_{f \in \mathcal{F}} |\{x : f(x) \neq g(x)\}| = \epsilon 2^n$.

Theorem 104. *The linearity test accepts linear functions $f : \{0,1\}^n \to \{0,1\}$ with certainty. If f is ϵ-far from linear, it rejects f with probability at least ϵ.*

We will prove Theorem 104 at the end. This theorem implies that, if the linearity test passes for each of f and g, the verifier can assume that they are close to linear functions $\tilde{f}$, $\tilde{g}$. However, they may still differ from $\tilde{f}$, $\tilde{g}$ on a small fraction of inputs. The following lemma shows that the verifier can in fact evaluate the linear functions $\tilde{f}$ and $\tilde{g}$ on arbitrary inputs.

Lemma 105. *If $f : \{0,1\}^n \to \{0,1\}$ is ϵ-far from some linear function $\tilde{f}$, $\tilde{f}(x)$ can be computed for any $x \in \{0,1\}^n$ with success probability $1 - 2\epsilon$ using 2 queries to f.*

Proof. In order to compute $\tilde{f}(x)$, pick $y \in \{0,1\}^n$ uniformly at random, and compute $f(x \oplus y) \oplus f(y)$. If f is linear (i.e. $f = \tilde{f}$), this equals $\tilde{f}(x)$. But if f is ϵ-far from linear, then

$$\Pr_y[f(x \oplus y) \oplus f(y) \neq \tilde{f}(x)] \leq \Pr_y[f(x \oplus y) \neq \tilde{f}(x \oplus y)] + \Pr_y[f(y) \neq \tilde{f}(y)] \leq 2\epsilon.$$

□

So, using the procedure of Lemma 105, the verifier can essentially assume that he has access to the linear functions $\tilde{f}$, $\tilde{g}$.

Lemma 106. *Given access to linear functions $\tilde{f} : \{0,1\}^n \to \{0,1\}$, $\tilde{g} : \{0,1\}^{n^2} \to \{0,1\}$, there is a test which uses two queries to $\tilde{f}$ and one to $\tilde{g}$ and such that if $\tilde{f} = L_u$ and $\tilde{g} = L_{uu^T}$ for some $u \in \{0,1\}^n$, the test accepts; otherwise, the test rejects with probability at least $1/4$.*

Proof. The test picks $r, s \in \{0,1\}^n$ uniformly at random and checks that $\tilde{f}(r)\tilde{f}(s) = \tilde{g}(rs^T)$. As $\tilde{f}$ is linear, we know that $\tilde{f} = L_u$ for some u. If $\tilde{g} = L_{uu^T}$, then

$$\tilde{g}(rs^T) = \bigoplus_{i,j=1}^{n} (uu^T)_{ij}(rs^T)_{ij} = \left(\bigoplus_{i=1}^{n} u_i r_i\right)\left(\bigoplus_{j=1}^{n} u_j s_j\right) = L_u(r)L_u(s) = \tilde{f}(r)\tilde{f}(s),$$

so the test passes. On the other hand, if $\tilde{f} = L_u$ but $\tilde{g} = L_B$, $B \neq uu^T$, then applying Lemma 103 to a row on which B and uu^T differ, we have

$$\Pr_s[Bs \neq (uu^T)s] \geq 1/2,$$

where the multiplication is done over $\mathbb{F}_2$. Thus

$$\Pr_{r,s}[\tilde{g}(rs^T) \neq \tilde{f}(r)\tilde{f}(s)] = \Pr_{r,s}[r^T Bs \neq r^T uu^T s] \geq 1/4,$$

which proves the claim. □

So, using the test of Lemma 106, the verifier can now assume that $\tilde{g} = L_{uu^T}$ for some $u \in \{0,1\}^n$. This implies that evaluating $\tilde{g}(A)$ for any $A \in \{0,1\}^{n^2}$ gives

$$\bigoplus_{i,j=1}^{n} A_{ij} u_i u_j.$$

The verifier is now finally able to apply this to check that u satisfies the system of equations S, using the following procedure. He picks $r \in \{0,1\}^m$ uniformly at random and calculates

$$\tilde{g}\left(\bigoplus_{k=1}^{m} r_k A^{(k)}\right) = \bigoplus_{k=1}^{m} r_k \bigoplus_{i,j=1}^{n} A_{ij}^{(k)} u_i u_j.$$

If the answer is $\bigoplus_{k=1}^{m} r_k b_k$, the verifier accepts; otherwise, he rejects. This test will always accept if u does indeed satisfy all the equations; if there is at least one equation u does not satisfy, it will accept with probability 1/2 (by Lemma 103). This completes the protocol.

Let us calculate the parameters of the entire protocol. First, it is clear that the verifier only examines a constant number of bits of the proof (14 bits!). If there is a u that satisfies all the equations in S, the prover can use this u throughout and the verifier will accept with certainty. If there is no such u, we check that the verifier rejects with constant probability. Assume the linearity test succeeds with probability at least 7/8 on each of f and g (otherwise, the verifier will reject with constant probability). This implies that each of these functions is 1/8-far from linear, so each computation of $\tilde{f}(x)$ or $\tilde{g}(x)$ for any x succeeds with probability at least 3/4. Thus, if $\tilde{g} \neq L_{uu^T}$ for some u, the test of Lemma 106 rejects with probability at least $(3/4)^3(1/4)$. On the other hand, if $\tilde{g} = L_{uu^T}$ for some u that does not satisfy every equation in S, the last step of the protocol will detect this with probability 1/2. So the verifier will reject incorrect "proofs" with at least constant probability; this probability can be boosted to 1/2 by repetition.

8.3.8 The linearity test

We still need to prove correctness of the linearity test (Theorem 104). To do so, we will use Fourier analysis on the group $\mathbb{Z}_2^n$, which we now briefly introduce.

For any positive integer n, recall that a character of the group $\mathbb{Z}_2^n$ is a function $\chi_S : \{0,1\}^n \to \{\pm 1\}$, $S \subseteq \{1,\ldots,n\}$, defined by $\chi_S(x) = (-1)^{\sum_{i \in S} x_i}$. Introduce an inner product on functions $f, g : \{0,1\}^n \to \mathbb{R}$, defined by

$$\langle f, g \rangle = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x) g(x).$$

We already proved the following (as Lemma 60).

Theorem 107 (Plancherel's theorem)**.** *The functions $\{\chi_S\}$ form an orthonormal basis with respect to the inner product $\langle \cdot, \cdot \rangle$.*

This implies that any function $f : \{0,1\}^n \to \mathbb{R}$ can be expanded in terms of characters, as follows:

$$f(x) = \sum_{S \subseteq [n]} \hat{f}(S)\chi_S(x),$$

for some real coefficients $\hat{f}(S)$, called the Fourier coefficients of f. These coefficients can be determined using the relation

$$\hat{f}(S) = \langle \chi_S, f \rangle = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{\sum_{i \in S} x_i} f(x).$$

Orthonormality further implies *Parseval's theorem*:

$$\frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)^2 = \langle f, f \rangle = \sum_{S \subseteq [n]} \hat{f}(S)^2.$$

We will want to apply Fourier analysis to boolean functions. It will be convenient to consider, instead of functions $f : \{0,1\}^n \to \{0,1\}$, functions $f : \{0,1\}^n \to \{\pm 1\}$ by mapping $0 \mapsto 1$, $1 \mapsto -1$ (we call this the ± 1 picture). In this picture, the linearity test previously discussed can be expressed as follows.

Definition 21 (Linearity test, ± 1 picture)**.** *Given access to a function $f : \{0,1\}^n \to \{\pm 1\}$, pick $x, y \in \{0,1\}^n$ at random and compute $f(x)$, $f(y)$ and $f(x \oplus y)$. Accept if $f(x)f(y)f(x \oplus y) = 1$; otherwise reject.*

Recall that our goal was to distinguish between the two cases of f being linear and far from linear. In the ± 1 picture, this is equivalent to distinguishing between f being a character χ_S, or far from any character.

If f is a character, it is clear that the linearity test always accepts. More generally, we can write down the probability that the linearity test accepts as follows.

Lemma 108. *The probability that the linearity test accepts a function $f : \{0,1\}^n \to \{\pm 1\}$ is*

$$\frac{1}{2} + \frac{1}{2} \sum_{S \subseteq [n]} \hat{f}(S)^3.$$

Proof. As we have $f(x)f(y)f(x \oplus y) \in \{\pm 1\}$, the probability that the test accepts is equal to the expectation of the indicator random variable $\mathrm{ACC}(x, y) = \frac{1}{2} + \frac{1}{2} f(x)f(y)f(x \oplus y)$. Thus

$$\Pr_{x,y}[\text{test accepts}] = \mathbb{E}_{x,y}[\mathrm{ACC}(x, y)] = \frac{1}{2} + \frac{1}{2}\mathbb{E}_{x,y}[f(x)f(y)f(x \oplus y)].$$

We now expand this last term in terms of the Fourier expansion of f, giving

$$\begin{aligned}
\mathbb{E}_{x,y}[f(x)f(y)f(x \oplus y)] &= \mathbb{E}_{x,y}\left[\left(\sum_{S \subseteq [n]} \hat{f}(S)\chi_S(x)\right)\left(\sum_{T \subseteq [n]} \hat{f}(T)\chi_T(y)\right)\left(\sum_{U \subseteq [n]} \hat{f}(U)\chi_U(x \oplus y)\right)\right] \\
&= \sum_{S,T,U \subseteq [n]} \hat{f}(S)\hat{f}(T)\hat{f}(U)\mathbb{E}_{x,y}[\chi_S(x)\chi_T(y)\chi_U(x \oplus y)].
\end{aligned}$$

Now observe that characters are "doubly" linear in the following sense:

$$\chi_S(x)\chi_T(x) = (-1)^{\sum_{i\in S} x_i}(-1)^{\sum_{j\in T} x_j} = (-1)^{\sum_{i\in S\Delta T} x_i} = \chi_{S\Delta T}(x),$$
$$\chi_S(x)\chi_S(y) = (-1)^{\sum_{i\in S} x_i + y_i} = \chi_S(x \oplus y).$$

Thus

$$\begin{aligned}\mathbb{E}_{x,y}[\chi_S(x)\chi_T(y)\chi_U(x\oplus y)] &= \mathbb{E}_{x,y}[\chi_S(x)\chi_T(y)\chi_U(x)\chi_U(y)] \\ &= \mathbb{E}_x[\chi_S(x)\chi_U(x)]\,\mathbb{E}_y[\chi_T(y)\chi_U(y)] \\ &= \mathbb{E}_x[\chi_{S\Delta U}(x)]\,\mathbb{E}_y[\chi_{T\Delta U}(y)] \\ &= \delta_{SU}\delta_{TU},\end{aligned}$$

where $\delta_{AB} = 1$ if $A = B$, and 0 otherwise, so

$$\mathbb{E}_{x,y}[f(x)f(y)f(x\oplus y)] = \sum_{S\subseteq[n]} \hat{f}(S)^3$$

as required. □

We can now complete the proof of Theorem 104, which we restate here in the ± 1 picture.

Theorem 109. *The linearity test accepts linear functions $f : \{0,1\}^n \to \{\pm 1\}$ with certainty. If f is ϵ-far from linear, it rejects f with probability at least ϵ.*

Proof. If f is linear, then $f = \chi_S$ for some $S \subseteq [n]$, so $\hat{f}(S) = 1$ and by Lemma 108 the linearity test passes with certainty. More generally, if f is ϵ-far from linear, then

$$1 - 2\epsilon = \max_{S\subseteq[n]} \langle f, \chi_S\rangle = \max_{S\subseteq[n]} \hat{f}(S).$$

By Lemma 108, the probability that the linearity test passes is

$$\frac{1}{2} + \frac{1}{2}\sum_{S\subseteq[n]} \hat{f}(S)^3 \le \frac{1}{2} + \frac{1}{2}\max_{S\subseteq[n]} \hat{f}(S) \sum_{T\subseteq[n]} \hat{f}(T)^2 = \frac{1}{2} + \frac{1}{2}\max_{S\subseteq[n]} \hat{f}(S) = 1 - \epsilon,$$

where the inequality uses the fact that $\hat{f}(S)^2 \ge 0$, and the first equality is Parseval's theorem. □

Finally, linearity is just one example of a property of boolean functions which is known to have an efficient tester. Oded Goldreich maintains a list of surveys about the now extensive field of property testing[9].

[9] `http://www.wisdom.weizmann.ac.il/~oded/test.html`