Complexity Theory

Notes for a course of lectures for students of Part 1B of the Computer Science Tripos.

Anuj Dawar
University of Cambridge Computer Laboratory
William Gates Building
JJ Thomson Avenue
Cambridge CB3 0FD.

Easter 2016

Texts

The main texts for the course are:

1. *Computational Complexity*. Christos H. Papadimitriou.
2. *Introduction to the Theory of Computation*. Michael Sipser.

Papadimitriou's textbook is now over twenty years old, but remains the standard textbook at this level. It covers everything we will do in this course and much more. The notation and concepts we follow in the course are mainly taken from here. Sipser's book is shorter but covers a wider range of material (including elements of computation theory, regular languages and grammars) but in less depth. It contains very accessible coverage of the topics covered in this course.

Other useful references include:

3. *Computers and Intractability: A guide to the theory of NP-completeness*. Michael R. Garey and David S. Johnson.
4. *P, NP and NP-completeness*. Oded Goldreich.
5. *Computability and Complexity from a Programming Perspective*. Neil Jones.

The book by Garey and Johnson is a classic. It was the first book on the subject of complexity theory when the topic first emerged in the 1970s. It is memorable for its long (and in the 1970s, comprehensive) list of known NP-complete problems which remains a valuable reference. The book by Goldreich is a recent book focussing on specifically on the complexity classes P, NP and the notion of NP-completeness. These topics are the central part of the present course (though not all of it), and the book provides an interesting personal perspective on them. The book by Jones provides an alternative view with an emphasis on complexity as it relates to programming, getting away from the abstract machine classes. It is a useful reference for the perspective it affords.

Course Outline

The following is a rough lecture-by-lecture guide, with relevant sections from the text by Papadimitriou (or Sipser, where marked with an S).

- **Algorithms and problems.** 1.1–1.3.
- **Time and space.** 2.1–2.5, 2.7.
- **Time Complexity classes.** 7.1, S7.2.
- **Nondeterminism.** 2.7, 9.1, S7.3.
- **NP-completeness.** 8.1–8.2, 9.2.

- **Graph-theoretic problems.** 9.3
- **Sets, numbers and scheduling.** 9.4
- **coNP.** 10.1–10.2.
- **Cryptographic complexity.** 12.1–12.2.
- **Space Complexity** 7.1, 7.3, S8.1.
- **Hierarchy** 7.2, S9.1.
- **Descriptive Complexity** 5.7, 8.3.

Algorithms and Problems

The aim of complexity theory is to understand what makes certain problems difficult to solve algorithmically. When we say that a problem is difficult, we mean not that it is hard to come up with an algorithm for solving a problem, but that any algorithm we can devise is inefficient, requiring inordinate amount of resources such as time and memory space. In this course, we aim at a theoretical understanding of why some problems appear to be inherently difficult.

The course builds upon **Algorithms**, where we saw how one measures the complexity of algorithms by asymptotic measures of the number of steps the algorithm takes. It also builds on **Computation Theory**, where we saw a formal, mathematical model of the concept of *algorithm*, which allows us to show that some problems (such as the Halting Problem) are not solvable at all algorithmically. We now wish to look at problems which are solvable (in the sense of computation theory) in that there is an algorithm, but they are not practically solvable because any algorithm is extremely inefficient. To start to make these notions precise, we look at a specific, familiar problem.

Sorting

Consider the statement.

> Insertion Sort runs in time $O(n^2)$.

This is shorthand for the following statement:

> If we count the number of steps performed by the Insertion Sort algorithm on an input of size n, taking the largest such number from among all inputs of that size, then the function of n so defined is *eventually* bounded by a *constant multiple* of n^2.

More formally, we define the notation O, Ω and θ as follows:

Definition

For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \leq cg(n)$;
- $f = \Omega(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \geq cg(n)$.
- $f = \theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

This allows us to compare algorithms. Thus, if we consider that the running time of Insertion Sort is also $\Omega(n^2)$, then the statement

> Merge Sort is an $O(n \log n)$ algorithm.

tells us that Merge Sort is an asymptotically faster algorithm than Insertion Sort. Whatever other constant factors might be involved, any implementation of the former will be faster than any implementation of the latter, for sufficiently large lists.

However, the question we are interested in is: what is the complexity of the sorting problem? That is, what is the running time complexity of the fastest possible algorithm for sorting a list? The analysis of Merge Sort tells us that this is no worse than $O(n \log n)$. In general, the complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem that the algorithm solves. To establish a *lower bound* we need to show that no possible algorithm, including those as yet undreamed of, can do better. This is a much harder task, in principle, and no good general purpose methods are known.

The case of the sorting problem is an exception, in that we can actually prove a lower bound of $\Omega(n \log n)$, showing that Merge Sort is asymptotically optimal.

Lower Bound. We will now establish the lower bound on the complexity of the sorting problem. That is, we aim to show that, for any algorithm A which sorts a list of numbers, the worst case running time of A on a list of n numbers is $\Omega(n \log n)$. We make no assumptions about the algorithm A other than it sorts a list of numbers, and makes its decisions on the basis of comparisons between numbers.

Suppose the algorithm sorts a list of n numbers $a_1, \ldots, a_n$. We can assume, without loss of generality, that the numbers are all distinct. Thus, whenever two numbers a_i and a_j are compared, the outcome is exactly one of the two possibilities: $a_i < a_j$ or $a_j < a_i$. When sorting the list of numbers $a_1, \ldots, a_n$, the algorithm A must reach its first decision point, which involves the comparison of two numbers a_i and a_j. Based on the result of this comparison, the algorithm will take one of two different branches, each of which may eventually lead to another comparison, and so on. Thus, the entire computation of the algorithm A can be represented as a tree (see Figure 1).

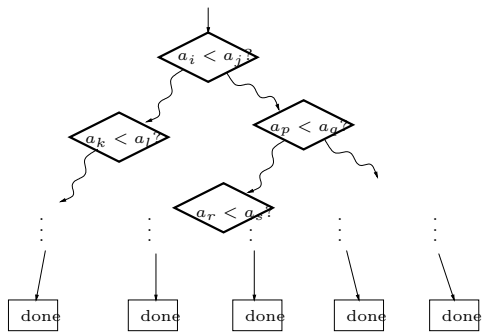

Figure 1: The computation tree for a sorting algorithm

The leaves of the tree represent a completed computation. How many leaves are there? We know that there are $n!$ different ways that the initial collection of n numbers could have been presented to the algorithm. Each of these must lead to a different branch through the computation tree. If this were not the case, we could find two different orderings of the same list, on which the algorithm performed exactly the same actions. Thus, the algorithm would incorrectly sort at least one of them. We conclude that the computation tree must have at least $n!$ leaves.

A binary tree with $n!$ leaves must have height at least $\log_2(n!)$. However, $\log(n!) = \theta(n \log n)$, establishing the result we wished to show. To see that $\log(n!) = \theta(n \log n)$, note the following inequalities:

$$\begin{aligned} \log(n!) = \log(n \cdot (n-1) \cdots 1) &= \log(n) + \log(n-1) + \cdots + \log(1) \\ &< \log(n) + \log(n) + \cdots + \log(n) = n \log n \end{aligned}$$

and

$$\begin{aligned} \log(n!) = \log(n \cdot (n-1) \cdots 1) &> \log(n) + \log(n-1) + \cdots + \log(n/2) \\ &> \log(n/2) + \log(n/2) + \cdots + \log(n/2) = (n/2) \log(n/2) \end{aligned}$$

The Travelling Salesman Problem The travelling salesman problem is defined as follows:

> Given a set V of vertices, along with a cost matrix, which is defined as a function $c : V \times V \to \mathbb{N}$, giving for each pair of vertices a positive integer cost, the problem is to find an ordering of the vertices $v_1, \ldots, v_n$ for which the total cost:
>
> $$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$
>
> is the smallest possible.

An obvious algorithm to solve the above problem is to try all possible orderings of the set of vertices, and compute the total cost for each one, finally choosing the one with the lowest cost. Since there are are $n!$ possible orderings, the running time of the algorithm is at least $n!$. There are somewhat better algorithms known, but they still have exponential time complexity.

In order to prove a lower bound on the complexity of the travelling salesman problem, we can carry out an analysis similar to the one given above for the sorting problem. That is, we can consider the computation tree of an arbitrary algorithm for solving this problem. Once again we see that there must be at least one branch in the computation tree corresponding to every possible ordering of the vertices. Thus, the computation tree must have at least $n!$ leaves and therefore the running time of the algorithm is $\Omega(n \log n)$. However, this is a far cry from the best known upper bound, which is $O(n^2 2^n)$. The gap between these two bounds sums up our state of knowledge (or lack of it) on the complexity of the travelling salesman problem. Indeed, it is emblematic of our state of knowledge of the complexity of many combinatorial problems.

Formalising Algorithms

As we have seen, one of the aims of complexity theory is to establish lower bounds on the complexity of problems, not just specific algorithms. To do this, we have to be able to say something about all algorithms for solving a particular problem. In order to say something

about all algorithms, it is useful to have a precise formal model of an algorithm. Fortunately, we have one at hand, introduced in Computation Theory for exactly this purpose. It is the Turing machine.

One important feature of the Turing machine as a formalisation of the notion of an algorithm is that it is simple. Algorithms are composed of very few possible moves or instructions. While this simplicity means it is not a formalism we would actually like to use to express algorithms, it does make proofs about *all* algorithms easier, as there are fewer cases to consider. By equivalence results we can prove, we can then be confident that the proofs apply equally well to other models of computation we can construct.

Turing Machines

For our purposes, a Turing Machine consists of:

- Q — a finite set of states;
- Σ — a finite set of symbols, disjoint from Q and including a distinguished blank symbol $\sqcup$ and a distinguished left-end marker $\triangleright$;
- $s \in Q$ — an initial state;
- $\delta : (Q \times \Sigma) \to Q \cup \{\texttt{acc}, \texttt{rej}\} \times \Sigma \times \{L, R, S\}$

  A transition function that specifies, for each state and symbol a next state (or accept `acc` or reject `rej`), a symbol to overwrite the current symbol, and a direction for the tape head to move (L – left, R – right, or S - stationary), with the proviso that for any q, $\delta(q, \triangleright) = (q', \triangleright, R)$ for some q'.

Informally, this is usually pictured as a box containing a finite state control, which can be in any of the states in Q, a tape (infinite in one direction), containing a finite string of symbols from Σ, and a read/write head pointing at a position in the tape. A complete snapshot of the machine at a moment in time can be captured if we know three things—the state of the control, the contents of the tape, and the position on the tape at which the head is pointing. We choose to represent this information as the following triple:

Definition

A *configuration* is a triple (q, w, u), where $q \in Q$ and $w, u \in \Sigma^\star$

The intuition is that (q, w, u) represents a machine in state q with the string wu on its tape, and the head pointing at the last symbol in w. The configuration of a machine determines its future behaviour.

Computation We think of a Turing machine as performing a computation by proceeding through a series of configurations. The transition from one configuration to the next is specified by the transition function δ. Formally,

> Given a machine $M = (Q, \Sigma, s, \delta)$ we say that a configuration (q, w, u) *yields in one step* (q', w', u'), written
>
> $$(q, w, u) \to_M (q', w', u')$$

if

- $w = va$;
- $\delta(q, a) = (q', b, D)$; and
- either $D = L$ and $w' = v\ u' = bu$
  or $D = S$ and $w' = vb$ and $u' = u$
  or $D = R$ and $w' = vbc$ and $u' = x$, where $u = cx$. If u is empty, then $w' = vb\sqcup$ and u' is empty.

The relation $\rightarrow_M^\star$ is the reflexive and transitive closure of $\rightarrow_M$. That is, for any two configurations c and c', we have $c \rightarrow_M^\star c'$ if the machine M can go from configuration c to configuration c' in 0 or more steps.

A *computation* of the machine M is a sequence of configurations $c_1, \ldots, c_n$ such that $c_i \rightarrow_M c_{i+1}$ for each i.

Each machine M defines a language $L(M) \subseteq \Sigma^\star$ which it accepts. This language is defined by:

$$L(M) = \{x \mid (s, \triangleright, x) \rightarrow_M^\star (\mathtt{acc}, w, u) \text{ for some } w \text{ and } u\}$$

Here, $\triangleright$ as a special symbol to denote the left end of the machine tape. By our definition, this symbol which cannot be overwritten by any other symbol, and when the head is reading $\triangleright$, it cannot move further to the left. We always assume that the string x does not contain any occurrences of $\triangleright$ or $\sqcup$.

So, $L(M)$ is the set of strings x such that if the machine M is started with the string x on the input tape, it will eventually reach the accepting state `acc`. Note that the strings excluded from $L(M)$ include those which cause the machine to reach the rejecting state as well as those which cause it to run forever. Sometimes it is useful to distinguish between these two cases. We will use the following definition:

> A machine M is said to *halt on input* x if there are w and u such that $(s, \triangleright, x) \rightarrow_M^\star (\text{acc}, w, u)$ or $(s, \triangleright, x) \rightarrow_M^\star (\text{rej}, w, u)$

We recall a few further definitions regarding languages that are accepted by some machine.

Definition

- A language $L \subseteq \Sigma^\star$ is *recursively enumerable* if it is $L(M)$ for some M.
- A language L is *decidable* if it is $L(M)$ for some machine M which *halts on every input*.
- A function $f : \Sigma^\star \rightarrow \Sigma^\star$ is *computable*, if there is a machine M, such that for all x, $(s, \triangleright, x) \rightarrow_M^\star (\mathtt{acc}, f(x), \varepsilon)$

where ε denotes the empty string. Recursively enumerable languages are also called *semi-decidable* as there is a machine that will confirm membership for any string that is in the language but may not halt on strings not in the language.

An example of a language that is recursively enumerable but not decidable is the *Halting Problem* H. To be precise, we fix a representation of Turing machines as strings over the alphabet $\{0,1\}$ (known as a Gödel numbering). If $[M]$ denotes the string representing a machine M, then Halting Problem H is defined as the language:

$$H = \{[M], x \mid M \text{ halts on input } x\}.$$

An example of a language that is not recursively enumerable is the complement of H, denoted $\bar{H}$:

$$\bar{H} = \{[M], x \mid M \text{ does not halt on input } x\}.$$

Example Consider the machine with δ given by:

	$\triangleright$	0	1	$\sqcup$
s	$s, \triangleright, R$	rej$, 0, S$	rej$, 1, S$	$q, \sqcup, R$
q	rej$, \triangleright, R$	$q, 1, R$	$q, 1, R$	$q', 0, R$
q'	rej$, \triangleright, R$	rej$, 0, S$	$q', 1, L$	acc$, \sqcup, S$

This machine, when started in configuration $(s, \triangleright, \sqcup 1^n 0)$ eventually halts in configuration $(\text{acc}, \triangleright \sqcup 1^{n+1} 0 \sqcup, \varepsilon)$. To see this, note that the machine, starting in state s, will move to the right and change state to q when it sees the blank. It will then leave all 1s unchanged as it continues to move to the right in state q, until it encounters the 0, which will be replaced with a 1. When it next encounters a blank, it changes state to q', changes the blank to 0 and moves again to the right, which, as the next symbol is again blank, triggers a transition to acc.

Our formal treatment of Turing machines can be extended easily to machines that have multiple tapes. We only give a brief indication here of how it would be done. For instance, if we have a machine with k tapes, we would specify it by:

- Q, Σ, s; and
- $\delta : (Q \times \Sigma^k) \to Q \cup \{\text{acc,rej}\} \times (\Sigma \times \{L, R, S\})^k$

That is, a transition is determined by the state and the k symbols that are under the k tape heads. Moreover, the transition determines the new state, the k symbols to be written on the tapes, and a direction for each tape head to move.

Similarly, to specify a configuration, we would need to specify the state, the contents of all k tapes, and the positions of all k tape heads. This can be captured by a $2k+1$ tuple as follows:

$$(q, w_1, u_1, \ldots, w_k, u_k).$$

It is known that any language that is accepted by a k tape Turing machine is also accepted by a one tape Turing machine.

Complexity

For any machine M which halts on all inputs, we define the *running time* of M to be the function $r : \mathbb{N} \to \mathbb{N}$ such that $r(n)$ is the length of the longest accepting computation of M on an input of length n.

For any function $f : \mathbb{N} \to \mathbb{N}$ we say that a language L is in $\mathsf{TIME}(f(n))$ if there is a machine M such that

- $L = L(M)$; and
- the running time of M is $O(f(n))$.

In short, $\mathsf{TIME}(f(n))$ is the set of all languages accepted by some machine with running time $O(f(n))$.

Similarly, we define $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses at most $O(f(n))$ tape cells on inputs of length n. In defining space complexity, we assume a machine M, which has a read-only input tape, and separate work tapes. We only count cells on the work tapes towards the complexity.

In general, not only can a single tape Turing machine simulate any other model of computation (multi-tape Turing machines, Random Access Machines, Java programs, the lambda calculus, etc.), but it can do so efficiently. That is, the simulation can be carried out by a Turing machine with only a polynomial factor increase in time and space complexity. This leads one to what is sometimes called the strong form of the Church-Turing thesis:

> Any two reasonable models of computation are polynomially equivalent, that is each can simulate the other within a polynomial factor of complexity.

There are, however, unreasonable models of computation, where it is not clear that a polynomial time simulation is possible. One such, for which we would dearly like to know whether or not it can be simulated with a polynomial time factor is the *nondeterministic* Turing machine.

Nondeterminism If, in the definition of a Turing machine, we relax the condition on δ being a function and instead allow an arbitrary relation, we obtain a *nondeterministic Turing machine*.

$$\delta \subseteq (Q \times \Sigma) \times (Q \cup \{\text{acc,rej}\} \times \Sigma \times \{R, L, S\}).$$

The notion of a configuration is unchanged. The state q, and the contents of the tape wu, with the position of the head given by the last symbol in w still give a snapshot of the machine. However, it is no longer the case that the configuration completely determines the future behaviour of the machine. More precisely, the yields relation $\to_M$ between configurations is no longer functional. For a given configuration (q, w, u), there may be more than one configuration (q', w', u') such that $(q, w, u) \to_M (q', w', u')$.

We still define the language accepted by M by:

$$L(M) = \{x \mid (s, \triangleright, x) \to_M^\star (\mathtt{acc}, w, u) \text{ for some } w \text{ and } u.\}$$

That is, it is the set of strings x for which there is some computation going from the starting configuration to an accepting configuration. It may, however, be the case that for some x in the language, there are other computations that lead to rejection, or that do not halt at all.

The computation of a nondeterministic machine, starting on a string x can be pictured as a tree of successive configurations.

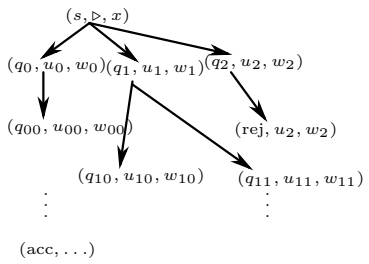

Figure 2: Nondeterministic computation

We say that the machine accepts the string x if there is any path in the tree that leads to an accepting configuration. Conversely, the machine does not accept x if all paths either lead to a rejecting state or are infinite.

A deterministic Turing machine can simulate a nondeterministic one, essentially by carrying out a breadth-first search of the computation tree, until an accepting configuration is found. Thus, for any nondeterministic machine M, there is a deterministic machine which accepts exactly the language $L(M)$. However, it is not clear that this simulation can be carried out in polynomial time. To say that a nondeterministic machine runs in polynomial time is to say that the height of the computation tree on input string x is bounded by a polynomial $p(|x|)$, in $|x|$—the length of x. However, the time required by a deterministic algorithm to carry out a breadth first search of a tree of height $p(|x|)$ is $O(2^{cp(|x|)})$, for some constant c.

Decidability and Complexity It is fairly straightforward to note that every decidable language has a time complexity in the sense that, if L is decidable, there is a *computable* function f such that $L \in \mathsf{TIME}(f(n))$. To see this, take a machine M such that $L = L(M)$ and such that M halts on all inputs, and take f to be the function that maps n to the maximum number of steps taken by M on any string $x \in L$ of length n. To see that f is computable, note that we can construct an algorithm which given an input number n, simulates M on all possible strings of length n. Since we know that M halts on all inputs, this process eventually terminates, and we can calculate the maximum number of steps taken by M on any of the inputs.

We could well ask if this might be true for a semi-decidable language. That is, suppose L is recursively enumerable but not decidable. So, there is a Turing machine M such that $L = L(M)$, though M might not halt on inputs not in L. So, there is still a well-defined function f which maps n to the maximum over all strings x in L of length n of the number of steps taken by M to accept x. However, we can show that f cannot be a computable function. We can say more: there is no computable function g such that $f = O(g)$. To see

why this is the case, suppose there were a computable function f such that for every $x \in L$ of length n, M accepts x in at most $f(n)$ steps. We can then construct a machine M' that accepts L and always halts, as follows. M', on input x, takes the length n of x and computes $f(n)$. M' then simulates M on input x for $f(n)$ steps, counting them along the way. If the simulation results in acceptance, M' accepts the input. However, if the simulation results in rejection, *or* the computation has not been completed in $f(n)$ steps, x is rejected. Thus, M' halts on all inputs, and accepts exactly the strings that M accepts, which is a contradiction, since we assumed that L is not decidable.

In other words, we have just shown that, for any semi-decidable language L, the running time of a machine M accepting L cannot be bounded above by any computable function.

Complexity Classes

A complexity class is a collection of languages. We determine a complexity class by specifying three things:

1. A model of computation (such as a deterministic Turing machine, or a nondeterministic Turing Machine, or a parallel Random Access Machine).
2. A resource (such as time, space or number of processors).
3. A set of bounds. This is a set of functions that are used to bound the amount of resource we can use.

What resources it is reasonable to consider depends on the model of computation we have chosen. We will, in general, consider Turing machines (either deterministic or nondeterministic). The resources we are primarily concerned with are *time* (which means the number of steps in a computation), and *space*, which means the maximum number of tape cells used in a computation. Many other models of computation allow similar notions of time and space to be defined. For instance, on a register machine, one could define space as the number of registers used, though a more reasonable measure might be to count the number of bits required to represent the numbers stored in the registers. Time could still be defined as the number of steps in the computation.

As long as we are considering deterministic machines, if we choose our set of functions (in item 3) to be broad enough, the languages that are included in the complexity class do not depend on the particular model of computation. So, if we take the collection of functions to be the set of all polynomials, it does not matter whether we consider Turing machines or register machines, we get the same class of languages either way. This is not the case if we take the class of all linear functions. So, whether or not a language is decidable in linear time is not just a property of the language, but is sensitive to the model of computation. However, it is reasonable to say that whether or not a language is decidable in polynomial time is a property of the language itself, much like the property of being decidable itself.

This leads us to the first important complexity class we will be studying:

$$\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k)$$

The class of languages decidable in polynomial time.

We identify this class of problems as those that we consider to be *feasibly* computable. That is, if we show that a particular language L is in this class, we say that L is *feasible* or *tractable*. Similarly, if we can prove that a language L is not in P, we say that L is *intractable*. One might argue whether or not this is a reasonable definition. After all, an algorithm that runs in time $\theta(n^{100})$ would hardly be regarded as efficient. On the other hand, there are algorithms whose worst-case running time is provably exponential but which are commonly used in practice and work quite well (for instance, the worst-case may simply not commonly arise).

Our reason for adopting polynomial time as our standard of feasibility is that it is a robustly and precisely defined class that lends itself to a thorough theoretical analysis. The development of a deep theory around this formal notion of feasibility gives us great insight into what makes certain problems more complex than others. This theoretical insight is then of great value in dealing with a large number of practical problems, and we would be deprived of the theory if we did not allow ourselves a simplifying definition of feasibility.

Example Problems

In this section we will look at a few classical decision problems with well-known algorithms. While the algorithms themselves are not difficult to analyse, our aim is to classify the complexity of the problem itself. The problems are chosen to represent a variety of different domains—graphs, numbers, Boolean expressions—and as we will see, they can stand in (in a very precise sense) for all members of a complexity class.

While we formally defined decision problems to be sets of *strings*, the example problems below may be sets of numbers or graphs or other kinds of objects. They can always be coded as sets of strings for formal purposes but at the informal level at which we define the algorithms, we leave out details of the coding.

Reachability

The Reachability problem is defined as the problem where, given as input a directed graph $G = (V, E)$, and two nodes $a, b \in V$ we are to decide whether there is a path from a to b in G. A straightforward algorithm for doing this searches through the graph, proceeding as follows:

1. mark node a, leaving other nodes unmarked, and initialise set S to $\{a\}$;
2. while S is not empty, choose node i in S: remove i from S and for all j such that there is an edge (i, j) and j is unmarked, mark j and add j to S;
3. if b is marked, accept else reject.

The algorithm as presented is somewhat vague in the details, but it can clearly be turned into a working implementation. To give a more detailed specification, one would have to state what data structure is used to implement S, and how the node i is chosen in step 2. For instance, S could be implemented as a stack, which would result in a depth-first search of the graph G, or it could be a queue, resulting in a breadth-first search. However, it should be reasonably clear that any implementation can be carried out on a Turing machine.

What is the time and space complexity of this algorithm? During the running of the algorithm, every edge in G must be examined at least once. Moreover, it can be seen that each edge is not examined more than once. This is because no vertex is added to S more than once, since once it is added, it is marked, and each edge is examined only when the vertex at its source is removed from S. So, we can safely say, if n is the number of vertices in the graph, that the running time of the algorithm is $O(n^2)$. An actual implementation on a Turing machine may require more time, but it can certainly be done in polynomial time, a point that has been emphasised several time earlier.

In terms of space, the only requirements for work space are the two sets—S and the set of marked vertices. Each can be implemented using n bits, one for each vertex. We may need some additional counters, each of $\log n$ bits, but the total work space requirement can be bounded by $O(n)$.

So, the above algorithm establishes that Reachability is a problem in P, and in SPACE$(O(n))$. However, the latter upper bound can be improved. As we shall see later, we can decide Reachability in less than linear space, at the expense of requiring extra time.

Euclid's Algorithm

Euclid's algorithm is the familiar algorithm for determining the greatest common divisor of two positive integers. But, it is an algorithm for computing a value rather than solving a decision problem. For our purposes, we will consider the decision problem (or language) defined as follows.

$$\mathsf{RelPrime} = \{(x, y) \mid \gcd(x, y) = 1\}.$$

The algorithm for solving it can be described as

1. Input (x, y).
2. Repeat until $y = 0$: $x \leftarrow x \bmod y$; Swap x and y
3. If $x = 1$ then accept else reject.

It is not difficult to see that after two iterations of Step 2, the value of x can be at most half of what it was before. Thus, the number of times the second step is repeated is at most $2 \log x$. Since the length of the input (in bits) is $\log x + \log y$, we conclude that RelPrime is in P. Note that if the algorithm took $\theta(x)$ steps to terminate, it would not be a polynomial time algorithm, as x is not polynomial in the *length* of the input.

Boolean Expressions

We define the collection of Boolean expressions to be the set of expressions formed from a given infinite set $X = \{x_1, x_2, \ldots\}$ of variables and the two constants `true` and `false` by means of the following rules:

- a constant or variable by itself is an expression;
- if ϕ is a Boolean expression, then so is $(\neg\phi)$;
- if ϕ and ψ are both Boolean expressions, then so are $(\phi \wedge \psi)$ and $(\phi \vee \psi)$.

If an expression contains no variables (that is, it is built up from just `true` and `false` using $\wedge$, $\vee$, and $\neg$), then it can be evaluated to either `true` or `false`. If an expression ϕ contains variables, then ϕ is not by itself true or false, Rather, we say that it is true or false for a *given* assignment of truth values. A *truth assignment* is just a function $T : X \to \{\texttt{true}, \texttt{false}\}$. We say that T *makes* ϕ *true* or T *satisfies* ϕ if, when we substitute $T(x)$ for each variable x in ϕ, we get an expression that evaluates to true. Conversely, if substituting $T(x)$ for each x in ϕ results in an expression that evaluates to `false`, we say that T *makes* ϕ *false* or T *does not satisfy* ϕ.
Examples

1. $(\texttt{true} \vee \texttt{false}) \wedge (\neg\texttt{false})$
   evaluates to `true`.
2. $(x_1 \vee \texttt{false}) \wedge ((\neg x_1) \vee x_2)$
   is satisfied by some truth assignments, but not by all.

3. $(x_1 \vee \texttt{false}) \wedge (\neg x_1)$

   is not satisfied by any truth assignment.

4. $(x_1 \vee (\neg x_1)) \wedge \texttt{true}$

   is satisfied by both possible truth assignments.

Evaluation With the definition of Boolean expressions established, we can begin to look at some algorithms for manipulating them. The first problem we look at is the evaluation problem. That is, given a Boolean expression *which does not contain any variables*, determine whether it evaluates to `true` or `false`. There is a deterministic Turing machine which can decide this, and which runs in time $O(n^2)$ on expressions of length n. The algorithm works by scanning the input, looking for subexpressions that match the left hand side of one of the rules below, and replacing it by the corresponding right hand side.

- $(\texttt{true} \vee \phi) \Rightarrow \texttt{true}$
- $(\phi \vee \texttt{true}) \Rightarrow \texttt{true}$
- $(\phi \vee \texttt{false}) \Rightarrow \phi$
- $(\texttt{false} \vee \phi) \Rightarrow \phi$
- $(\texttt{false} \wedge \phi) \Rightarrow \texttt{false}$
- $(\phi \wedge \texttt{false}) \Rightarrow \texttt{false}$
- $(\phi \wedge \texttt{true}) \Rightarrow \phi$
- $(\texttt{true} \wedge \phi) \Rightarrow \phi$
- $(\neg\texttt{true}) \Rightarrow \texttt{false}$
- $(\neg\texttt{false}) \Rightarrow \texttt{true}$

Each scan of the input, taking $O(n)$ steps must find at least one subexpression matching one of the rules. This can be formally proved by an induction on the structure of expressions, but should be intuitively clear. Since the application of a rule always removes at least one symbol from the expression, we can conclude that we cannot have more than n successive rule applications, as otherwise we would be left with no symbols at all. Thus, the algorithm takes $O(n^2)$ steps overall.

Circuit Value Problem

While Boolean expressions are a natural way to represent a Boolean function over a set of variables, a more compact representation is sometimes given by a *circuit*.

A circuit is a directed graph $G = (V, E)$, with $V = \{1, \ldots, n\}$ together with a labeling: $l : V \to \{\texttt{true}, \texttt{false}, \wedge, \vee, \neg\}$, satisfying:

- If there is an edge (i, j), then $i < j$;

- Every node in V has *indegree* at most 2.
- A node v has
  indegree 0 iff $l(v) \in \{\texttt{true}, \texttt{false}\}$;
  indegree 1 iff $l(v) = \neg$;
  indegree 2 iff $l(v) \in \{\vee, \wedge\}$

If we allow the nodes of indegree 0 to be labeled by variables from X, we obtain a way of representing arbitrary Boolean expressions. Moreover, the representation is more compact in that if the Boolean expression contains repeated subexpressions, they need not be repeated in the circuit, as we can take the output from a gate (or vertex in the directed graph) and use it as an input to several other gates.

For now, however, we concentrate on the case of circuits without variables and define the problem:

CVP—the *circuit value problem* is, given a circuit, determine the value of the result node n.

CVP is solvable in polynomial time, by the algorithm which examines the nodes in increasing order, assigning a value `true` or `false` to each node.

Prime Numbers

We now turn our attention to a particular familiar decision problem on numbers. It is simply this: given a number n, is it prime? Or, to formulate it as a language, it is

$$\mathsf{PRIME} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a prime number}\}.$$

There is an obvious algorithm for checking whether a number n is prime. For each integer $1 < m < \sqrt{n}$, check whether m divides n. This is not a polynomial time algorithm, since it requires $\sqrt{n}$ steps—which is exponential in the number of bits in n (remember we always measure complexity in terms of the size of the input, and the fair way to measure the size of a number is the number of bits it takes to represent it).

Consider also the language

$$\mathsf{COMP} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a composite number}\}$$

consisting of the binary representations of those numbers that are not prime. Is this language in P? It is easy to see that if COMP is in P, then so is PRIME. But, it is not obvious whether either of the languages is, in fact, in the class.

Satisfiability

While a Boolean expression without variables can simply be evaluated, for an expression that contains variables, we can ask different questions. One question is whether such an expression is *satisfiable*. That is, is there a truth assignment that makes it true?

We define SAT to be the set of all satisfiable Boolean expressions. This language can be decided by a deterministic Turing machine which runs in time $O(n^2 2^n)$. The straightforward

algorithm is to try all possible truth assignments. There are at most 2^n possible assignments, since there are at most n variables in the expression. For each one in turn, we can use the $O(n^2)$ algorithm described above to see if the resulting expression without variables is true.

A related problem is checking the validity of a Boolean expression. An expression is said to be *valid* if every possible assignment of truth values to its variables makes it true. We write VAL for the set of valid expressions. By an algorithm completely analogous to the one outlined for SAT above, we see that VAL is in time $O(n^2 2^n)$.

VAL and SAT are dual problems in the sense that a Boolean expression ϕ is valid if, and only if, the expression $\neg\phi$ is not satisfiable. This shows that an algorithm for deciding one language can easily be converted into an algorithm for deciding the other. No polynomial time algorithm is known for either problem, and it is widely believed that no such algorithm exists.

Hamiltonian Graphs

In a graph G with a set of vertices V, and a set of edges E, a *Hamiltonian cycle* is a path, starting and ending at the same vertex, such that every node in V appears on the cycle *exactly once*. A graph is called *Hamiltonian* if it contains a Hamiltonian cycle.[1] We define HAM to be the decision problem of determining whether a given graph is Hamiltonian. As an example, consider the two graphs in Figure 3. The first graph is not Hamiltonian, while the second one is.

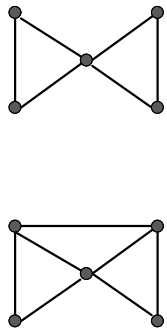

Figure 3: Which graph is Hamiltonian?

[1]The name comes from William Hamilton, a nineteenth century Irish mathematician, who considered whether there were ways of traversing the edges of platonic solids in such a way as to visit each corner exactly once.

Nondeterministic Polynomial Time

It is not difficult to establish that there is a nondeterministic algorithm that decides SAT in polynomial time. Indeed, time $O(n^2)$ suffices. The algorithm, in n nondeterministic steps, *guesses* a truth value for each variable. That is, for each variable in the expression, it makes a nondeterministic branch, where along one branch the variable is set to `true` and along the other it is set to `false`. Since the number of variables is no more than the length of the expression, n steps are sufficient for this process. Now, we have a Boolean expression ϕ and a truth assignment T for all its variables, so we can use the $O(n^2)$ deterministic algorithm described above to determine whether T satisfies ϕ. If there is any T that satisfies ϕ, some path through this nondeterministic computation will result in acceptance, so ϕ will be accepted.

Similarly, to determine whether a number x written in binary is in the language COMP, it suffices to *guess* a number y that is less than x and then check that y does divide x. The guessing can be implemented on a nondeterminisitic machine as a sequence of steps that each either writes a 0 or 1 on a tape, while the checking is easily done deterministically in polynomial time.

The problems COMP, SAT and HAM are all in the complexity class NP of problems that can be decided by a nondeterministic machine running in polynomial time. In general, a language in NP is characterised by a solution space which can be searched. For each possible input string x, there is a space of possible solutions to be searched. The number of such candidate solutions may be exponential in the length of x, but each solution can be represented by a string whose length is bounded by a polynomial p in the length of x. In particular, this means that the space can be searched by a backtracking search algorithm, where the depth of nesting of choice points is never greater than $p(|x|)$.

Another way of viewing the same class of problems is the generate and test paradigm. A language L in NP is one that can be solved by an algorithm with two components: a `Prover` and a `Verifier`. Given an input x, the `Prover` generates a proof V_x which demonstrates that x is in L. The `Verifier` then checks that the proof V_x is correct. As long as the length of V_x is bounded by a polynomial in the length of x *and* the `Verifier` runs in time polynomial in its input, the algorithm establishes that L is in NP (Fig. 4).

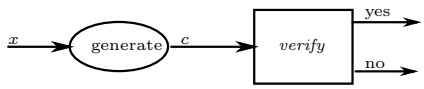

Figure 4: Generate and Test

In the case of COMP, the proof V_x can consist of a non-trivial factor of x, and the `Verifier` is a procedure that checks that V_x does indeed divide x, and is strictly between 1 and x. In the case of SAT, V_x could be an assignment of truth values to the variables of the Boolean formula x that makes it true, while in the case of HAM, V_x might be an enumeration of the vertices of the graph x in the order that they are visited by a Hamiltonian cycle.

Formally speaking, to define the class NP, we write NTIME(f) to denote the class of those languages L which are accepted by a *nondeterministic* Turing machine M, such that

for every $x \in L$ of length n, there is an accepting computation of M on x of length at most $f(n)$. Then

$$\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k)$$

Alternatively, say that a verifier V for a language L is a determnistic algorithm such that

$$L = \{x \mid (x, c) \text{ is accepted by } V \text{ for some } c\}$$

If V runs in time polynomial in the length of x, then we say that L is polynomially verifiable.

Now, it is not difficult to see that a language L is polynomially verifiable if, and only if, it is in NP. To see this, suppose L is a language which has a verifier V that runs in time $p(n)$ for some polynomial p. The following is then a description of a nondeterministic algorithm that accepts L.

1. input x of length n
2. nondeterministically guess c of length $\leq p(n)$
3. run V on (x, c)

Here, when we say "nondeterministically guess c" we mean that the algoritm writes a string of length $p(n)$ on the tape, and in doing so at each step there is a nondeterminsitic choice of which symbol to write on the tap. It is clear that for every possible string c of length at most $p(n)$, there is some computation sequence that results in c being written on the string.

In the other direction, suppose M is a nondeterministic machine that accepts L and runs in time $p(n)$. Assume further that in any configuration, there are at most k possible next configurations (for any machine M, there is a such a bound k). Now consider the *deterministic* algorithm V that takes as input (x, c) where c is a string of length $p(n)$ over a k-letter alphabet and simulates M as follows. At the i^{th} nondeterministic choice point, V looks at the i^{th} character in c to decide which branch to follow. If M accepts, then V accepts, otherwise it rejects. It can then be seen that V is a polynomial verifier for L.

SAT is typical of the problems in NP in the sense that the independent choices of truth assignments that can be made to the variables in a Boolean expression can encode the choices of any nondeterministic computation. It is in this sense that SAT "captures" nondeterministic time bounded computation. In the following, we make this notion of encoding nondeterministic choices precise.

Reductions

The idea of a *reduction* is central to the study of computability. It is used to establish undecidability of languages. Formally, given two languages $L_1 \subseteq \Sigma_1^\star$, and $L_2 \subseteq \Sigma_2^\star$, a reduction of L_1 to L_2 is a *computable* function $f : \Sigma_1^\star \to \Sigma_2^\star$, such that for every string $x \in \Sigma_1^\star$, $f(x) \in L_2$ if, and only if, $x \in L_1$.

In other words, every string in L_1 is mapped by f to a string in L_2, and every string that is not in L_1 is mapped to a string that is not in L_2. Note that there is no requirement for f

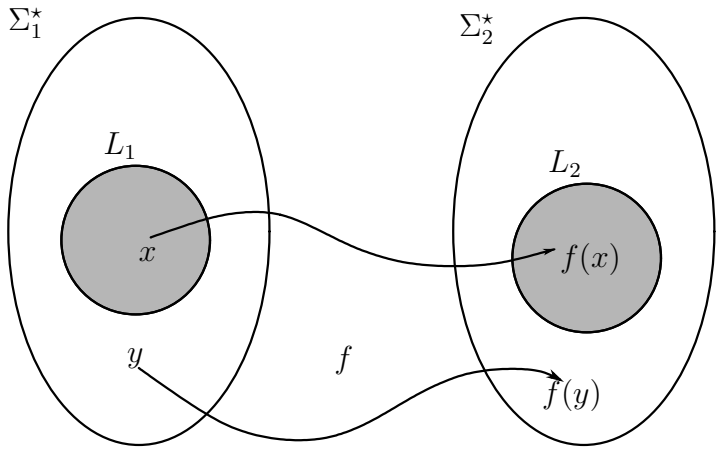

Figure 5: Reduction

to be surjective. In general, the range of f may be a small part of L_2, and a small part of its complement.

Reductions are useful in showing decidability (or, more usually, undecidability). If there is a reduction from a language L_1 to a language L_2, and L_2 is decidable, then we also know that L_1 is decidable. An algorithm to decide L_1 is obtained by computing, on any input x, $f(x)$ and then using the decision procedure for L_2 to determine a yes or no answer. If $f(x) \in L_2$, we know that $x \in L_1$ and if $f(x) \notin L_2$, we know that $x \notin L_1$. This argument is most useful in its contrapositive form: if there is a reduction from L_1 to L_2, and L_1 is known to be undecidable, then L_2 is undecidable as well. So, once we have proved one language (such as the Halting problem) to be undecidable, we can use reductions from it to prove other problems undecidable as well.

Resource Bounded Reductions When we are concerned about polynomial time computability rather than questions of decidability, we consider reductions that can be computed within bounded resources. To be precise, if f is a reduction from L_1 to L_2 and f is computable by an algorithm running in polynomial time, we say that L_1 is *polynomial time reducible* to L_2, which we write as

$$L_1 \leq_P L_2.$$

If $L_1 \leq_P L_2$, we can say that L_2 is at least as hard as L_1, at least in the sense of being polynomial time computable. In short, if $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$, then $L_1 \in \mathsf{P}$. This is for reasons analogous to those for decidability. We can compose the algorithm computing the reduction with the decision procedure for L_2 to get a polynomial time decision procedure for L_1. One point to be noted is that the string $f(x)$ produced by the reduction f on input x must be bounded in length by a polynomial in the length of x, since it is computed by a polynomial time algorithm. This is why the decision procedure for L_2 which runs in polynomial time on its input $f(x)$ is still running in time polynomial in the length of x.

NP-Completeness

The usefulness of reductions is in allowing us to make statements about the *relative complexity* of problems, even when we are not able to prove absolute lower bounds. So, even if we do not know whether or not there is a polynomial time algorithm for L_2, if $L_1 \leq_P L_2$, we can say if there were one, there would also be one for L_1. In this sense of relative complexity, Stephen Cook first showed that there are problems in NP that are maximally difficult.

Definition

A language L is said to be NP-hard *if for every language $A \in \mathsf{NP}$, $A \leq_P L$.*

A language L is NP-complete *if it is in* NP *and it is* NP*-hard.*

What Cook showed was that the language SAT of satisfiable Boolean expressions is NP-complete. In this sense, it is as hard as any problem in NP. A polynomial time algorithm for SAT would yield a polynomial time algorithm for every problem in NP.

We have already seen that SAT is in NP. To prove that it is NP-complete, we need to show that for every language L in NP, there is a polynomial time reduction from L To SAT. Since L is in NP, we know that there is a nondeterministic machine $M = (Q, \Sigma, s, \delta)$ and a polynomial p such that a string x is in L if, and only if, it is accepted by M within $p(|x|)$ steps. In what follows, we will assume, without loss of generality, that p is of the form n^k, where n is the length of x and k is a constant.

To establish the polynomial time reduction from L to SAT, we need to give, for each $x \in \Sigma^\star$, a Boolean expression $f(x)$ which is satisfiable if, and only if, there is an accepting computation of M on x. We construct $f(x)$ using the following variables:

$S_{i,q}$	for each $i \leq n^k$ and $q \in Q$
$T_{i,j,\sigma}$	for each $i, j \leq n^k$ and $\sigma \in \Sigma$
$H_{i,j}$	for each $i, j \leq n^k$.

The total number of variables is $|Q|n^k + |\Sigma|n^{2k} + n^{2k}$. The intended reading of these variables is that $S_{i,q}$ will be true if the machine at time i is in state q; $T_{i,j,\sigma}$ will be set to true if at time i, the symbol at position j in the tape is σ; and $H_{i,j}$ indicates that at time i, the head is pointing at position j on the tape.

Of course, these meanings are not inherent in the symbols. We have to construct the expression $f(x)$ to enforce them. The intention is that the only way to consistently assign truth values to these variables is to encode a possible computation of the machine M. The expression $f(x)$ is built up as the *conjunction* of the following expressions:

$$S_{1,s} \wedge H_{1,1} \tag{1}$$

which asserts that at the beginning, the state is s and the head is at the beginning of the tape.

$$\bigwedge_i \bigwedge_j (H_{i,j} \rightarrow \bigwedge_{j' \neq j} (\neg H_{i,j'})) \tag{2}$$

which asserts that the head is never in two places at once. That is, if $H_{i,j}$ is true for any i and j, then $H_{i,j'}$ must be false for any other j'.

$$\bigwedge_q \bigwedge_i (S_{i,q} \rightarrow \bigwedge_{q' \neq q} (\neg S_{i,q'})) \tag{3}$$

which asserts that the machine is never in two states at once. That is, for each state q and each state i, if $S_{i,q}$ is true, then $S_{i,q'}$ must be false for all other q'.

$$\bigwedge_i \bigwedge_j \bigwedge_\sigma (T_{i,j,\sigma} \rightarrow \bigwedge_{\sigma' \neq \sigma} (\neg T_{i,j,\sigma'})) \tag{4}$$

which asserts that each tape cell contains only one symbol. In other words, if $T_{i,j,\sigma}$ is true for any i, j and σ, then $T_{i,j,\sigma'}$ must be false for all other σ'.

$$\bigwedge_{j \leq n} T_{1,j,x_j} \wedge \bigwedge_{n<j} T_{1,j,\sqcup} \tag{5}$$

where x_j denotes the jth symbol in the string x. This expression asserts that at time 1, the tape contains the string x in its first n cells, and is blank after that.

$$\bigwedge_i \bigwedge_j \bigwedge_{j' \neq j} \bigwedge_\sigma (H_{i,j} \wedge T_{i,j',\sigma}) \rightarrow T_{i+1,j',\sigma} \tag{6}$$

which asserts that the tape only changes under the head. That is, if the head at time i is at position j, and at the same time position j' on the tape (for some other j') contains σ, then position j' still contains σ at time $i+1$.

$$\bigwedge_i \bigwedge_j \bigwedge_\sigma \bigwedge_q (H_{i,j} \wedge S_{i,q} \wedge T_{i,j,\sigma}) \rightarrow \bigvee_\Delta (H_{i+1,j'} \wedge S_{i+1,q'} \wedge T_{i+1,j,\sigma'}) \tag{7}$$

where Δ is the set of all triples (q', σ', D) such that $((q,\sigma),(q',\sigma',D)) \in \delta$ and

$$j' = \begin{cases} j & \text{if } D = S \\ j-1 & \text{if } D = L \\ j+1 & \text{if } D = R. \end{cases}$$

This asserts that the change from time step i to $i+1$, for each i is according to the transition relation δ. That is, if at time i, the head position is j, the state is q, and the symbol at position j is σ, then the state q' and head position j' at time $i+1$, as well as the symbol at position j at time $i+1$ are obtained by one of the possible transitions allowed by δ.

$$\bigvee_i S_{i,\text{acc}} \tag{8}$$

which asserts that at some time i, the accepting state is reached.

So, what has all this proved? We have shown that, for any nondeterministic machine M, any polynomial p and any input string x to M, we can write down a Boolean expression that is satisfiable if, and only if, M accepts x in time $p(|x|)$. What's more, in order to write down this Boolean expression, we don't actually have to run the machine M or perform any backtracking search. The expression simply encodes the description of M and x. It is therefore straightforward to check that the expression can be constructed in time polynomial in the length of x. The degree of the polynomial may depend on M and p.

Conjunctive Normal Form

A Boolean expression is in *conjunctive normal form* (or CNF) if it is the conjunction of a set of *clauses*, each of which is the disjunction of a set of *literals*, each of these being either a *variable* or the *negation* of a variable. Every Boolean expression is equivalent to one in CNF. In fact, any expression can be turned into an equivalent expression in CNF by repeated application of DeMorgan's laws, the laws of distributivity (of $\vee$ over $\wedge$) and by the law of double negation (which says that $\neg\neg\phi$ is equivalent to ϕ. There is, therefore, an algorithm for converting any Boolean expression into an equivalent expression in CNF. This is not, however, a polynomial time algorithm. We can prove that it requires exponential time (a rare example of a real lower bound result). This is because there are (for arbitrarily large n) expressions ϕ of length n such that the shortest CNF expression equivalent to ϕ has length $\Omega(2^n)$.

However, if we consider the reduction constructed above from any language L in NP to SAT, and take the Boolean expressions that result from the reduction, then there is a polynomial time algorithm that will convert them into equivalent CNF expressions. This is because the formulas are almost in CNF already. In particular, the expression is a conjunction of expressions, of which (1) is just a conjunction of literals (and is therefore in CNF). The expressions in (2), (3) and (4) can be easily converted into CNF by distributing the implication over the innermost conjunction. For example, (2) can be rewritten as

$$\bigwedge_i \bigwedge_j \bigwedge_{j' \neq j} (H_{i,j} \rightarrow (\neg H_{i,j'})).$$

This is now in CNF (recall that $\phi \rightarrow \psi$ is just shorthand for $\neg\phi \vee \psi$). Note also that while (2) contains only one occurrence of the variable $H_{i,j}$ for each i and j, the CNF version above has $n^k - 1$ occurrences of each $H_{i,j}$. This is however, a fixed polynomial increase. Similarly, (5) and (6) are already in CNF. The expression (7) requires a bit more work, but it is not too difficult, and is left here as an exercise.

We can conclude that, for each language L in NP, there is, in fact, a polynomial time computable function f such that $f(x)$ is a CNF expression for all x, and $f(x)$ is satisfiable if, and only if, $x \in L$. In other words, if we define CNF-SAT to be the collection of all satisfiable CNF expressions, then we can say that we have shown that CNF-SAT is NP-complete.

We define a further restriction on our expressions. A Boolean expression ϕ is said to be in 3CNF if it is in CNF, i.e. $\phi \equiv C_1 \wedge \ldots \wedge C_m$, and each clause C_i is the disjunction of no more than 3 literals. We also define 3SAT to be the set of those expressions in 3CNF that are satisfiable.

While it is not the case that every Boolean expression is equivalent to one in 3CNF, what we can say is that for every CNF expression ϕ, there is an expression ϕ' in 3CNF so that ϕ' is satisfiable if, and only if, ϕ is. Moreover, there is an algorithm, running in polynomial time, which will convert ϕ to ϕ'. We will illustrate this with an example. Suppose we have a clause C with four literals

$$C \equiv (l_1 \vee l_2 \vee l_3 \vee l_4).$$

Introducing new variables n_1 and n_2, we can write down an expression ψ in 3CNF

$$\psi \equiv (l_1 \vee l_2 \vee n_1) \wedge (\neg n_1 \vee l_3 \vee n_2) \wedge (\neg n_2 \vee l_4).$$

This expression is not equivalent to C because, for one thing, it contains variables that are not in C. But, ψ is satisfiable if, and only if, C is. The idea can be easily generalised to clauses with any number of variables. Moreover, we can verify that the number of new variables and clauses introduced is no more than the number of literals in the clause being replaced. This ensures that the conversion can be carried out in polynomial time.

What we can conclude from this is that there is a polynomial time computable reduction from CNF-SAT to 3SAT. This can be combined with the fact that CNF-SAT is NP-complete to show that 3SAT is also NP-complete.

NP-Complete Problems

The argument at the end of the last section, which takes us from the NP-completeness of CNF-SAT to the NP-completeness of 3SAT can be formalised in terms of the composition of reductions. That is, if for any languages L_1, L_2 and L_3, we have that $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then it must be the case that $L_1 \leq_P L_3$. In other words, the relation $\leq_P$ between languages is transitive. The reason is that if f is a reduction from L_1 to L_2, and g a reduction from L_2 to L_3, then their composition $g \circ f$ is a reduction from L_1 to L_3. Moreover, if both f and g are polynomial time computable, then so is their composition, by the algorithm that first computes f on its input x and then computes g on the result $f(x)$. There is a polynomial bound on the running time of this algorithm because the length of $f(x)$ must be bounded by a polynomial in the length of x, so even though the input to the algorithm computing g is $f(x)$, the total running time is bounded by a polynomial in the length of x.

By the transitivity of reducibility, and our previous proofs of NP-completeness, it follows that if we show, for any language A in NP, that SAT $\leq_P A$ or that 3SAT $\leq_P A$, it immediately follows that A is NP-complete. We now use this to establish the NP-completeness of a number of natural combinatorial problems.

Graph Problems

We begin by looking at problems involving graphs.

Independent Set Let $G = (V, E)$ be an undirected graph with a set V of vertices, and E of edges. We say that $X \subseteq V$ is an *independent set* if there are no edges (u, v) in E, for any $u, v \in X$. The definition gives rise to a natural algorithmic problem, namely, given a graph G, find the largest independent set. Instead of this optimisation problem, we will consider a decision problem which we call IND, which is defined as

> The set of pairs (G, K), where G is a graph, and K is an integer, such that G contains an independent set with K or more vertices.

That is, we turn the question into a yes/no question by explicitly setting a target size in the input.

The problem IND is clearly in NP. We can nondeterministically generate an arbitrary subset X of the vertices, and then in polynomial time check that X has at least K elements and that it is an independent set.

To show that IND is NP-complete, we construct a reduction from 3SAT to IND. The reduction maps a Boolean expression ϕ in 3CNF with m clauses to the pair (G, m) where G is a graph, and m the target size. G is obtained from the expression ϕ as follows.

> G contains m triangles, one for each clause of ϕ, with each node representing one of the literals in the clause (for clauses containing fewer than three literals, we can label more than one node in the triangle with the same literal).
>
> Additionally, there is an edge between two nodes in different triangles if they represent literals where one is the negation of the other.

As an example, if ϕ is the expression

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

we obtain a graph G with six nodes, connected by edges as in Figure 6, where the triangle of vertices at the top corresponds to the first clause and the triangle at the bottom to the second clause of ϕ.

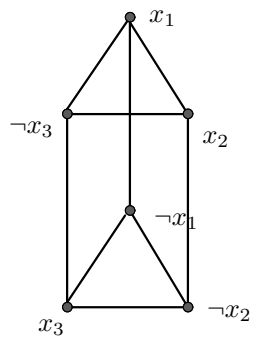

Figure 6: Reduction from 3SAT to IND

To prove that this is a polynomial time reduction from 3SAT to IND, we need to show that the transformation from ϕ to (G, m) can be carried out by a polynomial time algorithm. This is not too difficult to see, as G is really just a direct encoding of ϕ as a graph, and m is obtained by counting the clauses of ϕ. We also need to check that G contains an independent set of m vertices *if, and only if,* ϕ is satisfiable.

For one direction, suppose ϕ is satisfiable, and let T be a truth assignment that satisfies it. For each clause in ϕ, choose exactly one literal in the clause which is made true by T (there must be at least one in each clause, since T satisfies ϕ). Let X be the set of vertices in G corresponding to the literals we have just chosen. X cannot contain a vertex labeled by a variable x and another vertex labeled by $\neg x$, otherwise T would not be a consistent truth assignment. Also, X cannot contain two vertices from the same triangle, since we chose exactly one vertex in each clause. So, there are no edges between vertices in X. Furthermore, since we chose one vertex from each triangle, there are m vertices in X.

In the other direction, we have to show that if G has an independent set with m vertices, then ϕ is satisfiable. Let X be such an independent set. Any two vertices arising from the same clause are part of a triangle, and so cannot be both in X. So, X must contain exactly one vertex from each triangle. We now define a truth assignment T for ϕ as follows. For any variable x, if there is a vertex labeled x in X, then let $T(x) = \texttt{true}$, and if there is a vertex labeled $\neg x$ in X, then let $T(x) = \texttt{false}$. If X contains neither a vertex labeled x nor a vertex labeled $\neg x$, then set $T(x)$ arbitrarily. We can see that this is a consistent assignment of truth values to the variables of ϕ, since X cannot contain both a vertex labeled x and a vertex labeled $\neg x$ as there would be an edge between them. Finally, we note that T is a truth assignment that satisfies ϕ, since for each clause of ϕ, there is one literal corresponding to a vertex in X, and which is therefore made true by T.

Clique A graph problem closely related to IND, and perhaps more commonly mentioned, is the problem of finding a *clique* in a graph. Once again, we begin with a definition. Given a graph $G = (V, E)$, a subset $X \subseteq V$ of the vertices is called a *clique*, if for every $u, v \in X$, (u, v) is an edge.

Once again, there is a natural optimisation problem of finding the largest clique in a graph, but we will consider a decision problem, which we call CLIQUE.

> The set of pairs (G, K), where G is a graph, and K is an integer, such that G contains a clique with K or more vertices.

As with IND, it is easy to see that CLIQUE is in NP. There is an algorithm which, on input (G, K) guesses a subset X of the vertices of G containing K elements, and then verifies that X forms a clique. To see that CLIQUE is NP-complete, it suffices to prove that IND $\leq_P$ CLIQUE. This is easily seen by the reduction that maps a pair (G, K) to the pair $(\bar{G}, K)$, where $\bar{G}$ is the *complement graph* of G. That is, $\bar{G}$ has the same set of vertices as G, and a pair (x, y) is an edge of $\bar{G}$ if, and only if, it is not an edge of G. Clearly then, any independent set of G is a clique in $\bar{G}$, and conversely any clique in $\bar{G}$ is an independent set of G, so G contains an independent set with K elements if, and only if, $\bar{G}$ contains a clique with K elements. The reduction can, quite obviously, be carried out by a polynomial time algorithm.

Graph Colourability If we are given a graph $G = (V, E)$, that is a set of vertices V along with a set of edges E, we call a *colouring* of G an assignment of colours to the vertices of G such that no edge connects two vertices of the same colour. We say that G is k-colourable if there is a colouring of G which uses no more than k colours. More formally, we say that G is *k-colourable*, if there is a function

$$\chi : V \to \{1, \ldots, k\}$$

such that, for each $u, v \in V$, if $(u, v) \in E$,

$$\chi(u) \neq \chi(v).$$

This sets up a decision problem for each k. Namely,

> given a graph $G = (V, E)$, is it k-colourable?

The problem 2-Colourability is in P. However, for all $k > 2$, k-colourability is NP-complete. Note that here, unlike in the cases of IND and CLIQUE considered above, the number k is not part of the input presented for the algorithm. Rather, we are considering k as a number fixed before hand and the input is just a graph. So, we will show that the problem 3-colourability is NP-complete. The problem is clearly in NP, since we can guess a colour (one of a fixed set of three, say `red, blue, green`) for each vertex, and then verify that the colouring is valid by checking for each edge that its endpoints are differently coloured. The checking can be done in polynomial time, so this algorithm establishes that 3-colourability is in NP.

To complete the proof that 3-colourability is NP-complete, we construct a reduction from 3SAT to 3-colourability. The reduction maps a Boolean expression ϕ in 3-CNF to a graph G so that G is 3-colourable if, and only if, ϕ is satisfiable. Suppose ϕ has m clauses and n distinct variables. G will have 2 special vertices, which we call a and b, one vertex for each variable x and one for its negation $\bar{x}$. For each x, the vertices a, x and $\bar{x}$ are connected in a triangle. In addition, there is an edge connecting a and b, and for each clause in ϕ, there are five new vertices connected in the pattern shown in Figure 7, with the vertex b, and the vertices corresponding to the three literals l_1, l_2 and l_3 that appear in the clause.

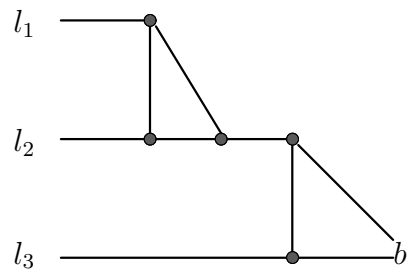

Figure 7: Gadget for the reduction from 3SAT to 3-Colourability

We now need to check that G is 3-colourable if, and only if, ϕ is satisfiable. In any colouring of G, the vertex a must be assigned a colour, let's call it `red`. Since there is an edge between a and b, b must have a different colour, let's call it `blue`. Furthermore, the vertices corresponding to the literals must be coloured `blue` or `green`, as they all have edges to a. Also, of these two colours, each vertex x must be of the opposite colour as its negation $\bar{x}$. We now claim that if there is a valid colouring of the whole graph G, then the truth assignment that makes a variable x `true` if, and only if, it is coloured `blue` (i.e. the same colour as b) is a satisfying truth assignment of ϕ. Conversely, from any satisfying truth assignment T of ϕ, we can obtain a valid 3-colouring of G by colouring `blue` all vertices corresponding to literals that are made `true` by T and colouring `green` all literals that are made `false` by T. To see that this is the case, we only need to check that, in the gadget shown in Figure 7, if b is `blue`, and l_1, l_2, l_3 are all either `blue` or `green`, then there is a valid colouring of the remaining vertices if, and only if, at least one of l_1, l_2 or l_3 is `blue`. This can be checked by examining all possibilities, and is left as an exercise.

Hamiltonian Graphs Recall the problem HAM of determining whether a given graph G contains a Hamiltonian cycle. It is not too difficult to see that HAM is a problem in NP. An algorithm for solving it can guess a permutation $v_1, v_2, \ldots, v_n$ of the vertices in V, and then verify that, for each i, there is an edge from v_i to v_{i+1} and finally that there is an edge from v_n to v_1. Since the verification step can be done by a polynomial time algorithm, we conclude that HAM is in NP.

We can show that HAM is NP-hard by a reduction from 3SAT. This involves coding a Boolean expression ϕ as a graph G in such a way that every satisfying truth assignment of ϕ corresponds to a Hamiltonian cycle in G. The reduction is much more involved then the

ones we have seen for IND and 3-Colourability. The details are left out of the present notes, and can be found in standard textbooks on Complexity Theory.

Travelling Salesman Problem Recall that the *Travelling Salesman Problem* is an optimisation problem specified as, given

- V — a set of vertices.
- $c : V \times V \to \mathbb{N}$ — a cost matrix.

Find an ordering $v_1, \ldots, v_n$ of V for which the total cost:

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

is the smallest possible.

In order to analyse this with the theory of NP-completeness we are building up, we can turn this too into a decision problem, by putting in an explicit target t for the cost of the tour. That is, the problem TSP is the set of triples $(V, c : V \times V \to \mathbb{N}, t)$, such that there is an ordering $v_1, \ldots, v_n$ of V for which

$$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1}) \leq t.$$

It is fairly easy to see that if there were a polynomial time solution to the optimisation problem, there would also be a polynomial time solution to the decision problem TSP. We could just compute the optimal solution and then check whether its total cost was within the budget t. Thus, a proof that TSP is NP-complete is a strong indication that there is no polynomial time solution to the optimisation problem.

To show that TSP is NP-hard, we note that there is a simple reduction to it from HAM. The reduction maps a graph $G = (V, E)$ to the triple $(V, c : V \times V \to \mathbb{N}, n)$, where n is the number of vertices in V, and the cost matrix c is given by:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \textit{otherwise} \end{cases}$$

Now, since a tour must visit all n cities, it must traverse exactly n edges in this matrix. Thus, if it is within budget, i.e. it has a total cost at most n, then it must only use entries with value 1 in the matrix. In other words, it only traverses edges in the original graph G and, therefore, describes a Hamiltonian cycle in the graph. Conversely, if there is a Hamiltonian cycle in G, there is a way of touring all the cities using only edges of cost 1, and this gives a tour of total cost n.

Sets, Numbers and Scheduling

So far, the NP-complete problems we have looked at either concern the satisfiability of formulas, or properties of graphs. However, NP-completeness is not about formulas and graphs. Literally thousands of naturally-arising problems have been proved NP-complete, in areas involving network design, scheduling, optimisation, data storage and retrieval, artificial intelligence and many others, and new ones are found every day. Such problems arise naturally whenever we have to construct a solution within constraints, and the most effective way appears to be an exhaustive search of an exponential solution space. In this section, we examine three more NP-complete problems, whose significance lies in that they have been used to prove a large number of other problems NP-complete, through reductions. They have been chosen as representative of a large class of problems dealing with sets, numbers and schedules.

3D Matching 3D matching is an extension into 3 dimensions of the well known bipartite matching problem. The latter is defined as the problem of determining, given two sets B and G of equal size, and a set $M \subseteq B \times G$ of pairs, whether there is a *matching*, i.e. a subset $M' \subseteq M$ such that each element of B and each element of G each appear in exactly one pair M' (note that this implies that M' has exactly n elements). The bipartite matching problem is solvable by a polynomial time algorithm.

The problem *3D Matching*, also sometimes called *tripartite matching* is defined by:

> Given three disjoint sets X, Y and Z, and a set of triples $M \subseteq X \times Y \times Z$, does M contain a matching?
>
> I.e. is there a subset $M' \subseteq M$, such that each element of X, Y and Z appears in exactly one triple of M'?

This problem is NP-complete. We prove the NP-hardness by a reduction from 3SAT.

We are given a Boolean expression ϕ in 3CNF with m clauses and n variables. For each variable v, we include in the set X, m distinct elements $x_{v1}, \ldots, x_{vm}$ and in Y also m elements $y_{v1}, \ldots, y_{vm}$. We also include in Z $2m$ elements for each variable v. We call these elements $z_{v1}, \ldots, z_{vm}, \bar{z}_{v1}, \ldots, \bar{z}_{vm}$. The triples we include in M are (x_{vi}, y_{vi}, z_{vi}) and $(x_{vi}, y_{v(i+1)}, \bar{z}_{vi})$ for each $i < m$. In the case where $i = m$, we put 1 instead of $i + 1$ in the last triple. The situation for $m = 4$ is displayed in Figure 8, where the triangles represent triples of M.

In addition, for each clause c of ϕ, we have two elements $x_c \in X$ and $y_c \in Y$. These elements are additional to the m elements for each variable mentioned earlier. If, for some variable v, the literal v occurs in clause c, we include the triple

$$(x_c, y_c, z_{vc})$$

in M, while if the literal $\neg v$ occurs in c, we include the triple

$$(x_c, y_c, \bar{z}_{vc})$$

in M.

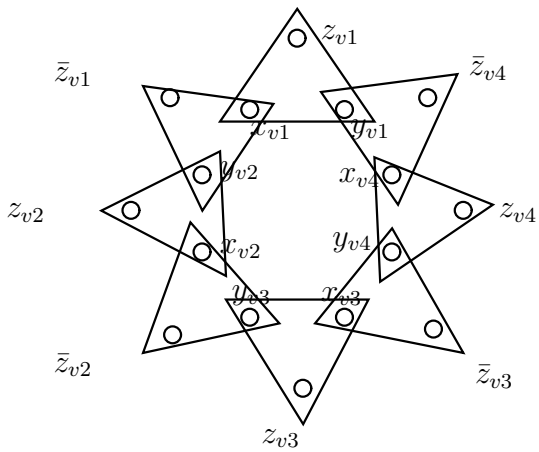

Figure 8: Matching gadget for each variable

Note that, so far, the sets X and Y each contain $mn + m$ elements (m elements x_{vi} for each variable v, and one element x_c for each clause c), and the set Z contains $2mn$ elements, $2m$ for each variable. To get a valid instance of 3DM we need three sets of equal size. We achieve this by adding $m(n-1)$ additional elements to X and Y. These are *dummy* elements, in the sense that we don't want them to constrain possible matchings in any way, so for each dummy x and y, and *every* element of z, we include the triple (x, y, z) in M. The result is that if we can find a matching for all elements except the dummies, we can find a matching for the whole set.

It remains to show that there is a matching for the instance we have constructed if, and only if, the expression ϕ is satisfiable. Note that, for any variable v, the element x_{v1} of X appears in only two triples: (x_{v1}, y_{v1}, z_{v1}) and $(x_{v1}, y_{v2}, \bar{z}_{v1})$. A successful matching must include exactly one of these. Moreover, once we have made the choice, this constrains the choice for all other x_{vi}, as can be seen from Figure 8. In fact, there are only two ways that all the x_{vi} can be matched, for a particular v. We can either use all the z_{vi} or we can use all the $\bar{z}_{vi}$. We can think of this choice as the two possible truth values that the variable v can take.

Consider any truth assignment T satisfying the expression ϕ. For each variable v that is set to `true` by T, we select all triples of the form $(x_{vi}, y_{v(i+1)}, \bar{z}_{vi})$ for our matching, and for each v that is set to false, we select the triples $(x_{vi}, y_{v(i)}, z_{vi})$. The result is that, if v is `true`, the elements z_{vi} are available to satisfy x_c and y_c for clauses c in which v appears as a positive literal. Similarly, if v is `false`, the elements $\bar{z}_{vi}$ are available to satisfy those clauses where $\neg v$ appears as a literal. Thus, from a satisfying truth assignment, we obtain a matching. Conversely, we can argue that any matching yields a satisfying truth assignment. The details of the argument are left as an exercise.

Set Covering A further two well-known NP-complete problems are established by straightforward reductions from 3DM. The first is *Exact Cover by 3-Sets* which is defined by:

Given a set U with $3n$ elements, and a collection $S = \{S_1, \ldots, S_m\}$ of three-

element subsets of U, is there a sub collection containing exactly n of these sets whose union is all of U?

The straightforward reduction maps an instance (X, Y, Z, M) of 3DM to the pair (U, S), where $U = X \cup Y \cup Z$, and S consists of all the three element sets $\{x, y, z\}$, where $(x, y, z) \in M$.

A more general problem is *Set Covering*, which is defined by:

> Given a set U, a collection of $S = \{S_1, \ldots, S_m\}$ subsets of U and an integer budget B, is there a collection of B sets in S whose union is U?[2]

The reduction from *Exact Cover by 3-Sets* to *Set Covering* maps a pair (U, S) to the triple (U, S, n), where $3n$ is the number of elements in U.

Knapsack Knapsack is one of the most famous NP-complete problems because it is a natural generalisation of many scheduling and optimisation problems, and through a variety of reductions has been used to show many such problems NP-hard. The optimisation problems we have seen so far all involve attempting to either minimize some measure of cost or maximize some quantitative benefit. Many optimisation problems arising in practice, however, involve tradeoffs between cost and benefit. Knapsack captures this intuition by involving both a maximisation and a minimisation element. Formally, the problem is defined by:

> We are given n items, each with a positive integer value v_i and weight w_i. We are also given a maximum total weight W, and a minimum total value V.
>
> Can we select a subset of the items whose total weight does not exceed W, and whose total value exceeds V?

To prove that Knapsack is NP-complete, we construct a reduction from the problem of *Exact Cover by 3 Sets* (we omit the argument that Knapsack is in NP, which is easy).

We are given a set $U = \{1, \ldots, 3n\}$, and a collection of 3-element subsets of U, $S = \{S_1, \ldots, S_m\}$. We map this to an instance of KNAPSACK with m elements each corresponding to one of the S_i, and having weight and value

$$\Sigma_{j \in S_i}(m+1)^{j-1}$$

and set the target weight and value both to

$$\Sigma_{j=0}^{3n-1}(m+1)^j.$$

The idea is that we represent subsets of U as strings of 0s and 1s of length $3n$. We treat these strings as representations of numbers, not in base 2, but in base $m+1$. This guarantees that when we add numbers corresponding to the sets S_i, we never get carry from one place to the next, as there are only m sets. Thus, the only way we can achieve the target number (represented by 1s in all positions), is if the union of the sets we have chosen is all of U, and

[2]The use of an integer budget B in the definition of the problem is a clear indication that this is the decision version of a natural optimisation problem.

no element of U is represented more than once—as this would result in a value greater than 1 in some place. It follows that the instance of Knapsack has a solution if, and only if, the original pair (U, S) has an exact cover by 3-sets.

Indeed, the reduction we have constructed produces instances of Knapsack of a rather special kind. All weights and values are equal, and the target weight is the same as the target value. While this does prove that the general Knapsack problem is NP-complete, it also establishes the NP-completeness of a restriction of the problem to instances where weights and values are always equal. This problem has a particularly simple formulation:

> Given a collection of numbers $v_1, \ldots, v_n$ and a target t, is there a subcollection of the numbers which adds up exactly to t?

This simple looking problem turns out to be NP-complete.

Scheduling The problem Knapsack has been used to prove a wide variety of scheduling problems NP-complete. A few examples are given here as illustration.

Timetable Design

> Given a set H of *work periods*, a set W of *workers* each with an associated subset of H (available periods), a set T of *tasks* and an assignment $r : W \times T \to \mathbb{N}$ of *required work*, is there a mapping $f : W \times T \times H \to \{0, 1\}$ which completes all tasks?
>
> That is, for any $w \in W$ and $h \in H$ there is at most one $t \in T$ for which $f(w, h, t) = 1$, and there is one only if w is available at h. Moreover, for each $t \in T$, and each $w \in W$, there are at least $r(w, t)$ distinct h for which $f(w, h, t) = 1$.

Sequencing with Deadlines

> Given a set T of *tasks* and for each task a *length* $l \in \mathbb{N}$, a release time $r \in \mathbb{N}$ and a deadline $d \in \mathbb{N}$, is there a work schedule which completes each task between its release time and its deadline?
>
> Here, a schedule is an assignment to each task $t \in T$ a start time $s(t)$, such that $s \geq r(t)$, $d(t) \geq s(t) + l(t)$, and such that for any other t', $s(t') \geq s(t) + l(t)$.[3]

Finally, a multi-processor version of this is:

Job Scheduling

> Given a set T of *tasks*, a number $m \in \mathbb{N}$ of processors a length $l \in \mathbb{N}$ for each task, and an overall deadline $D \in \mathbb{N}$, is there a multi-processor schedule which completes all tasks by the deadline?

[3]Note that this is non-preemptive scheduling.

Responses to NP-completeness

Having seen a number of NP-complete problems, and some varied proofs of their NP-completeness, we have acquired at least some ability to recognise new NP-complete problems when we see them. One question that arises is, what are we to do when we are confronted with one. Surely, the analysis that shows that a problem is NP-complete is not the end of the matter. We are still required to find a solution of some sort. There are a variety of possible responses.

It might be that we are trying to solve a single instance—we might have to construct a railway timetable, or an exam timetable, once only. The results of asymptotic complexity do not, of course, tell us much about single instances. An algorithm whose running time is exponential may run in reasonable time on instances up to some small size. In practice, though, an algorithm that is exponential in running time will become completely impractical on even reasonably small instances. One would certainly not want to use it on something as large as a railway timetable. The running time may, after all, double with the addition of each additional station

In general, if we are using a general purpose algorithm for the problem (rather than exploiting features of the particular single instance), then *scalability* is important. A program tested on small instances may completely seize up when confronted with an industrial scale example.

Thus, in order to get a solution that is scalable, one has to adopt a different approach to a brute force search. Often, this involves a closer examination of the problem at hand. Is it really an NP-complete problem in its full generality, or is it a *restriction* to some special class of instances? For instance, in many applications, the graphs that arise are necessarily planar. Not all the NP-complete graph problems we have considered remain NP-complete when restricted to planar graphs. The CLIQUE problem is a case in point. No planar graph can have a clique of five or more elements. Thus, the problem of finding the largest clique in a graph reduces to checking whether the graph contains a clique of four elements, something that can be done by a polynomial time algorithm. While HAM and 3-Colourability are known to be NP-complete, even when restricted to planar graphs, 4-Colourability is trivial on planar graphs, since all planar graphs are 4-colourable.

Another approach often adopted in dealing with optimisation problems corresponding to NP-complete problems is to settle for approximate rather than exact solutions. Often, such problems admit polynomial time *approximation algorithms*, which are not guaranteed to find the best solution, but will produce a solution which is known to be within a known factor of the optimal. This is particularly useful in applications where we need to be able to give performance guarantees.

A final point is that, if we are using an algorithm with potentially exponential worst-case performance, using a backtracking search strategy, it is important to identify good *heuristics* for constraining the search. These heuristics will often arise from known limitations of the actual application area, and it is difficult to devise general purpose rules for them. However, good heuristics can dramatically cut down the search space and home in on a solution for a typical instance of the problem, while still requiring an exponential search in the worst-case.

Certificates, Function Classes and Cryptography

We originally defined NP as the collection of languages accepted by a nondeterministic machine in polynomial time. We have also used another equivalent notion, that of polynomial verifiability. This is essentially the definition of NP we have been using in establishing that various problems are in the class. We now look at a formal statement that characterises it in this way. A language $L \subseteq \Sigma^\star$ is in NP if, and only if, it can be expressed as:

$$L = \{x \mid \exists y R(x, y)\},$$

where R is a relation on strings satisfying two conditions:

1. R is decidable in polynomial time by a deterministic machine.
2. R is *polynomially balanced*. That is, there is a polynomial p such that if $R(x, y)$ and the length of x is n, then the length of y is no more than $p(n)$.

In such a case, if $R(x, y)$ holds, we say that y is a certificate of the membership of x in L. Or, as we might have said earlier, it is a "solution" to x. For example, in the case where L is SAT and x is a Boolean expression, y would be an assignment of truth values to the variables of x and $R(x, y)$ would be the relation that holds if y makes x true. Similarly, in the case of 3-colourability, x would be a graph, y an assignment of colours to the graph and $R(x, y)$ would hold if y was a valid colouring.

co-NP

We have seen a number of examples of languages in NP and, in each case, established this fact by exhibiting a suitable notion f a certificate. Here is another candidate language. Recall that the set VAL is the set of Boolean expressions that are valid, i.e. true for any assignment of values to the variables. VAL and SAT are dual problems in the sense that a Boolean expression ϕ is valid if, and only if, the expression $\neg\phi$ is not satisfiable. This shows that a *deterministic* algorithm for deciding one language can easily be converted into an algorithm for deciding the other. So, just as with SAT, VAL is in $\mathsf{TIME}(n^2 2^n)$. But, is VAL in NP? What would a certificate consist in for showing that $\phi \in \mathsf{VAL}$?

Consider, instead the language $\overline{\mathsf{VAL}} = \{\phi \mid \phi \notin \mathsf{VAL}\}$, the *complement* of VAL, i.e. the set of Boolean expressions that are not valid. This language is clearly in NP. To check that ϕ is not valid, it suffices to guess a *falsifying* truth assignment and verify it. Such an algorithm does not work for VAL. In this case, we have to determine whether *every* truth assignment results in `true`—a requirement that does not sit as well with the definition of acceptance by a nondeterministic machine.

If we interchange accepting and rejecting states in a deterministic machine that accepts the language L, we get one that accepts $\overline{L}$. If a language $L \in \mathsf{P}$, then also $\overline{L} \in \mathsf{P}$. Complexity classes defined in terms of nondeterministic machine models are not necessarily closed under complementation of languages.

The class of languages co-NP is defined as the complements of languages in NP. So, if L is in co-NP, there is a polynomial time decidable and polynomially balanced relation S

such that $\bar{L} = \{x \mid \exists y S(x,y)\}$. In other words $L = \{x \mid \forall y \neg S(x,y)\}$. Since P is closed under complementation, $\neg S(x,y)$ is decidable in polynomial time. So, we can equivalently characterise the class co-NP as the set of languages L for which there is a polynomial-time decidable relation R such that:

$$L = \{x \mid \forall y\, |y| < p(|x|) \rightarrow R(x,y)\}.$$

This situation is often summed up by saying that NP is the collection of those languages for which there are succinct certificates of membership and co-NP is the collection of languages for which there are succinct certificates of disqualification. Another way of saying this is that if NP is the class of problems that are polynomially verifiable, than co-NP is the class of problem that are *polynomially falsifiable*.

Since every language in P is also in NP, and the complement of a language in P is itself in P, it follows that every language in P is also in co-NP. In other words $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{co\text{-}NP}$. The situation is depicted in figure 9.

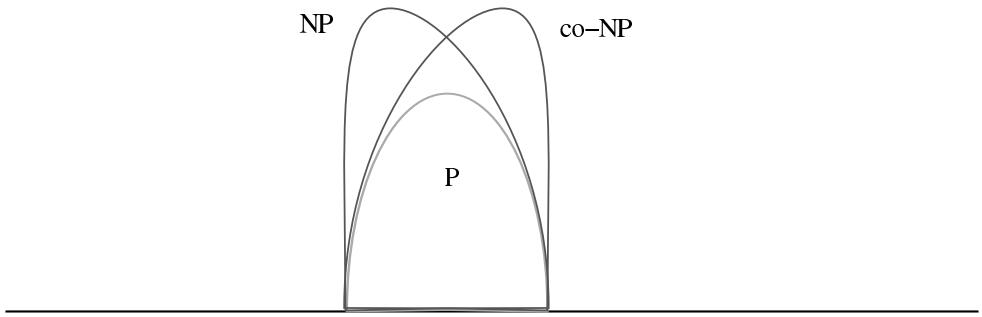

Figure 9: Relation between the classes P, NP and co-NP

Clearly, if it were the case that $\mathsf{P} = \mathsf{NP}$, it would follow that $\mathsf{NP} = \mathsf{co\text{-}NP}$. Other than that, we know nothing about whether the containments pictured in figure 9 are proper. Any of the following situations is consistent with what we can prove so far:

- $\mathsf{P} = \mathsf{NP} = \mathsf{co\text{-}NP}$
- $\mathsf{P} = \mathsf{NP} \cap \mathsf{co\text{-}NP} \neq \mathsf{NP} \neq \mathsf{co\text{-}NP}$
- $\mathsf{P} \neq \mathsf{NP} \cap \mathsf{co\text{-}NP} = \mathsf{NP} = \mathsf{co\text{-}NP}$
- $\mathsf{P} \neq \mathsf{NP} \cap \mathsf{co\text{-}NP} \neq \mathsf{NP} \neq \mathsf{co\text{-}NP}$

Validity There is a notion of completeness for co-NP entirely analogous to that of NP-completeness.

Definition A language L is said to be *co-NP-complete* if L is in co-NP and for every language $A \in \mathsf{co\text{-}NP}$, $A \leq_P L$.

Indeed, the complement of any NP-complete language is co-NP-complete. This is because if f is a a reduction from a language L_1 to a language L_2 then it is also a reduction of $\bar{L_1}$–the

complement of L_1–to $\bar{L_2}$–the complement of L_2, as $x \in \bar{L_1} \Leftrightarrow x \notin L_1 \Leftrightarrow f(x) \notin L_2 \Leftrightarrow f(x) \in \bar{L_2}$. So, $\overline{\mathsf{SAT}}$ is co-NP-complete.

There is an easy reduction from $\overline{\mathsf{SAT}}$ to VAL. It is just the function that maps a Boolean expression ϕ to $\neg\phi$. It follows that VAL is co-NP-complete. Hence, if there were a polynomial time algorithm for solving VAL, this would imply that P = NP = co-NP. Similarly, if we could show that VAL is in NP it would follow that every problem in co-NP is in NP by composing the polynomial time reduction of the problem to VAL with the nondeterministic machine deciding VAL.

Prime Numbers Recall our earlier discussion of the languages

$$\mathsf{PRIME} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a prime number}\}$$

and

$$\mathsf{COMP} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a composite number}\}$$

which are complements of each other. The straightforward algorithm for determining whether a number n is prime is to search for a factor of n in the range $1 \ldots \sqrt{n}$. This actually shows that COMP is in NP, and hence PRIME is in co-NP, since the factor is a succinct certificate of membership in COMP.

However, Vaughn Pratt showed in 1976 that PRIME is in fact in NP. Unlike other problems we have seen in NP, it is not immediately obvious that this is the case. Indeed, the succinct certificates of primality that Pratt demonstrated exist are quite involved to describe. Astoundingly, in August 2002, Manindra Agrawal and his students Neeraj Kayal and Nitin Saxena announced that PRIME is actually in P! The algorithm they present is quite easy to describe but the proof that its running time is bounded by a polynomial relies on deep facts from number theory about the distribution of certain kinds of primes.

To be precise, the algorithm by Agrawal et al. embeds the problem of checking primality into that of factoring polynomials. If a and p are co-prime integers, then the univariate polynomial $(x-a)^p$ is equivalent to $x^p - a$ modulo p if, and only if, p is a prime. That is to say, if p is prime, then all monomials in the expansion of $(x-a)^p$ except the first and last vanish and if p is not prime, then at least one of them does not vanish. Of course, to check the equivalence

$$(x-a)^p \equiv (x^p - a) \pmod{p}$$

could require exponential time. However, Agrawal et al. show that it is sufficient to check it *modulo* a polynomial $x^r - 1$ for suitable small values of r. What suitable means is spelled out precisely in their paper, and it is then shown that the existence of such suitable values follows from some deep results in number theory.

Function Classes

So far, in discussing complexity of problems we have only considered decision problems—those where there is a yes/no answer for every possible input. In some cases, as with the travelling salesman problem, this was somewhat artificial. We forced what is naturally

an optimisation problem into the mould of a decision problem in order to make it fit our theory. The same could be said of the problems Clique, IND and a variety of the scheduling problems we considered. The justification for this is that we were concerned with establishing lower bounds. And, if we could prove that there is no polynomial time algorithm for the decision problem TSP, we would know that there is none for the optimisation problem either. Similarly, the proof of NP-completeness of TSP shows that if there were a polynomial-time algorithm for the optimisation problem, then P = NP. Furthermore, there is a connection in the other direction as well. A polynomial-time algorithm for the decision problem could also be used to obtain a polynomial-time algorithm for the optimisation problem. This is true for problems (TSP, Clique and IND are among them) where the maximum possible value of the measure being optimised can be represented by a string that is at most polynomial in the length of the input. This is because a black-box solving the decision problem could be used by a binary search algorithm to find the optimum value. For instance, given a black box that given a pair G, K gives a yes/no answer to the question whether G has a clique with at least K elements, we can find out the size of the largest clique in G with $\log |G|$ queries to the black box.[4]

Still, there is something interesting one can say about the complexity of *functions* as opposed to decision problems and one can develop a complexity theory around such things. While we know what the function computed by a deterministic machine is, it is not easy to speak of the function computed by a nondeterministic machine since the string that is produced as output for a given input x is not determined. Instead, we will talk of the complexity of the witness functions of languages in NP.

Suppose we have a language L in NP. Then, we know

$$L = \{x \mid \exists y R(x, y)\}$$

where R is a polynomially-balanced, polynomial time decidable relation.

Definition

A *witness function* for L is any function f such that:

- if $x \in L$, then $f(x) = y$ for some y such that $R(x, y)$;
- $f(x) =$ "no" otherwise.

The class FNP is the collection of all witness functions for languages in NP.

It is reasonably clear that if an NP-complete problem L had a polynomial-time computable witness function, then P would be the same as NP. For instance, a witness function for SAT would be a function which, for any Boolean expression ϕ returns a satisfying truth assignment for ϕ if there is one and the string "no" otherwise.

Conversely, if P = NP, then every language in NP has a witness function that is computable in polynomial time, by a binary search algorithm. To be precise, if $L = \{x \mid \exists y R(x, y)\}$ is a language in NP, then the set $L' = \{(x, z) \mid \exists y < z R(x, y)\}$ is also decidable in NP, where $<$ is the lexicographical order on strings. So, assuming P = NP, there is a polynomial time algorithm for L' and we can use repeated calls to this algorithm to find, given x, a witness y using binary search.

[4] Actually finding a clique is a different matter.

We could also define a notion of polynomial-time reduction between functions under which one can show that there is a witness function for SAT that is FNP-complete. Still, there are interesting functions in FNP, beside those that are associated with NP-complete problems. We look at one such next.

Factorisation The factorisation problem is that of finding, given a positive integer n, its prime factors. Since we also want to be able to verify that a given factorisation is correct, we define the factorisation function as the function that maps n to a tuple

$$(2, k_1; 3, k_2; \ldots; p_m, k_m),$$

where $n = 2^{k_1} 3^{k_2} \cdots p_m^{k_m}$.

This function is in FNP because it is a witness function for a trivial problem in NP, namely the set of all positive integers. Still, it is not known whether this function is computable in polynomial time. Indeed, public key cryptographic systems are built on the assumption that factorisation cannot be done quickly.

Cryptography

The fundamental aim of cryptography is to enable Alice and Bob to communicate (as in figure 10) without Eve being able to eavesdrop.

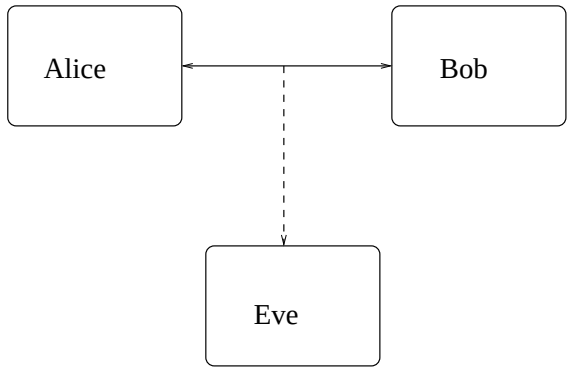

Figure 10: Alice, Bob and Eve

In any cryptographic system, there are two keys: e – the encryption key and d – the decryption key. There are also two functions D and E such that: for any x, $D(E(x, e), d) = x$. A private key system relies for its security on keeping both e and d secret.

For instance, we could take $d = e$ and let both D and E be the *exclusive or* function. This is known as the one-time pad. Since $(x \oplus e) \oplus e = x$, this satisfies the required condition for a cryptographic system. Furthermore, the one time pad is provably secure in that the only way that Eve would be able to decode a message is by knowing the key. For, if Eve knows both the plain text x and the encrypted text y, she can work out the key $e = x \oplus y$.

In contrast, in a public key cryptographic system the key e is made public while d is kept secret. We still have functions D and E such that: for any x, $D(E(x, e), d) = x$. If

the system is to work in reasonable time, then both D and E should be computable in polynomial time. However, if Eve is not to be able to eavesdrop, the function that maps $E(x, e)$ to x without knowing d should not be computable in polynomial time. However, this function is necessarily in the class FNP. This is because it is a witness function for the set:

$$\{y \mid \exists x E(x, e) = y\}.$$

Thus, public key cryptography is not provably secure in the way that the one-time pad is. Instead, it relies on the unproved hypothesis that there are functions in FNP that are not in FP (the class of functions computable in polynomial time).

One Way Functions To be more precise, in order for public-key cryptography to work, we would like to have a *one-way function*, which is defined as a function f meeting the following conditions:

1. f is one-to-one.
2. for each x, $|x|^{1/k} \leq |f(x)| \leq |x|^k$ for some k.
3. $f \in \mathsf{FP}$.
4. $f^{-1} \notin \mathsf{FP}$.

The reason for the first condition is obvious: we don't want to have two distinct plain text strings map to the same encrypted string. The second condition ensures that neither the map from plain text to encrypted text nor the reverse map will result in a greater than polynomial increase in the length of the string. The third condition is to ensure that encryption can be done efficiently and the last is to ensure that decryption is difficult.

To be really useful, we would also like f to satisfy the condition that it can be easily inverted when supplied with a decryption key d as a parameter. Note, however, that we don't even know whether there are any functions that satisfy the four conditions above. Indeed to prove that such one-way functions exists would involve proving that $\mathsf{P} \neq \mathsf{NP}$. However, a great deal of money has been invested in the belief that the RSA function, defined by:

$$f(x, e, p, q) = (x^e \bmod pq, pq, e)$$

is a one-way function. This takes the plain text x and an encryption key e along with two primes p and q and produces the encrypted text $x^e \bmod pq$ along with the public key (pq, e).

Given our inability to prove lower bounds, we are unable to prove that RSA is secure. It's reasonable to ask, however, whether we can place as much confidence in it as in our belief that P and NP are different. Can we prove that, assuming $\mathsf{P} \neq \mathsf{NP}$, RSA is not invertible in polynomial time? Indeed, can we show, under the assumption that $\mathsf{P} \neq \mathsf{NP}$ that some one-way function exists? Is there a function f that satisfies the first three conditions in the definition of a one-way function and for which f^{-1} is FNP-complete? The answer to these questions is, unfortunately, no. The existence of one-way functions is logically equivalent to a stronger statement than $\mathsf{P} \neq \mathsf{NP}$. To understand this, we need to introduce one further complexity class.

Definition
A nondeterministic machine is *unambiguous* if, for any input x, there is at most one accepting computation of the machine.

UP is the class of languages accepted by unambiguous machines in polynomial time.

Equivalently, we could say that UP is the class of languages L for which:

$$\{x \mid \exists y R(x, y)\},$$

where R is polynomial time computable, polynomially balanced, *and* for each x, there is *at most one* y such that $R(x, y)$. In other words, R is a partial function.

It is clear from the definitions that $\mathsf{P} \subseteq \mathsf{UP} \subseteq \mathsf{NP}$. It is considered unlikely that there are any NP-complete problems in UP. It is, in fact, difficult to think of any natural problems that are in UP but not in P. However, we can show that the existence of one-way functions is equivalent to the statement that $\mathsf{P} \neq \mathsf{UP}$.

To see why this is the case, first assume that we have a one-way function f. Then, we can show that the language L_f defined by

$$L_f = \{(x, y) \mid \exists z (z \leq x \text{ and } f(z) = y\}.$$

Here, we are assuming that f acts on positive integers represented by binary strings. It is not too difficult to see that L_f is in UP, since there is a nondeterministic machine that recognises it by "guessing" a value for z and then checking that $f(z) = y$. Since the function f is one-to-one, there is at most one such z and therefore the machine is unambiguous.

To see that L_f is not in P, we note that if it was, then we could compute f^{-1} in polynomial time, using a binary search procedure. That is, given a value y, we know that if there is a z such that $f(z) = y$, then it has $z \leq 2^{(\log y)^k}$, by condition 2 in the definition of a one-way function. So, we can find z using a binary search procedure making at most $(\log y)^k$ calls to the algorithm that checks for membership in L_f. This gives a polynomial time algorithm for computing f^{-1}, contradicting the assumption that f is one-way.

We have shown that the existence of a one-way function implies that $\mathsf{P} \neq \mathsf{UP}$. Conversely, suppose that we have a language L that is in UP but not in P and U is an unambiguous machine accepting L. Define the function f_U by the following:

> if x is a string that encodes an accepting computation of U, then $f_U(x) = 1y$ where y is the input string accepted by this computation;
>
> $f_U(x) = 0x$ otherwise.

The function f_U is one-to-one because the machine is unambiguous and therefore a given string y cannot have more than one accepting computation. It is polynomially bounded by the fact that machine U runs in polynomial time and therefore the lengths of computations are bounded in the length of the input. It is also easily checked that f_U is in FP and if f_U^{-1} were in FP, then the language L would be in P.

Space Complexity

Recall that we defined $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses at most $O(f(n))$ tape cells on inputs of length n. In defining space complexity, we assume a machine M, which has a read-only input tape, and separate work tapes. We only count cells on the work tapes towards the complexity. Similarly, we define $\mathsf{NSPACE}(f(n))$ to be the class of languages accepted by a *nondeterministic* machine which uses at most $O(f(n))$ tape cells on inputs of length n. Since we are only counting space on the work tape, it makes sense to consider bounding functions f that are sub-linear—it is quite possible for a machine to accept a string x while using work space that is less than the length of x.

Constructible Functions When choosing a function f to serve as a bound of resources, for example in defining a complexity class such $\mathsf{SPACE}(f(n))$, we need to be careful. For one thing, as we saw before, it makes sense to only consider computable functions. However, there are some quite unnatural computable functions—for instance, a function that is 2^n for even n, and $\log n$ for odd numbers, which will naturally lead to quite unnatural classes of languages if used in the definition of a complexity class. From now on, we restrict the functions we use for bounds to what are called *constructible* functions.

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *constructible* if:

- f is non-decreasing, i.e. $f(n+1) \geq f(n)$ for all n; and
- there is a deterministic machine M which, on any input of length n, replaces the input with the string $0^{f(n)}$, and M runs in time $O(n + f(n))$ and uses $O(f(n))$ *work space* (recall we do not count space on the input tape).

The intuition behind the second requirement is that computing the function f shouldn't require more resources than the limit imposed by f itself. This will, in particular, allow us to compose the computation of f with any other computation that takes place within $O(f(n))$ time and space. Thus, we can prove results such as the following:

> If L is in $\mathsf{TIME}(f(n))$, then there is a machine M that accepts L, and which halts on all inputs in $O(f(n))$ steps.

Examples All of the following functions are constructible:

- $\lceil \log n \rceil$;
- n^2;
- n;
- 2^n.

If f and g are constructible functions, then so are $f + g$, $f \cdot g$, 2^f and $f \circ g$ (this last, provided that $f(n) > n$). This, together with the above examples, allows us to generate all the constructible functions we will ever need, including all polynomials $p(n)$, all functions $2^{p(n)}$ for polynomials p, and much else besides.

Space Complexity Classes

As with time complexity, we are not generally interested in complexity classes defined by single functions. We consider wider classes, in order to obtain robust definitions of complexity classes that are independent of particular machine models. The classes we are particularly interested in are the following:

$\mathsf{L} = \mathsf{SPACE}(\log n)$ The class of languages decidable using logarithmic workspace.

$\mathsf{NL} = \mathsf{NSPACE}(\log n)$ The class of languages recognisable by a nondeterministic machine using logarithmic workspace.

$\mathsf{PSPACE} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k)$
The class of languages decidable in polynomial space.

$\mathsf{NPSPACE} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k)$ The class of languages recognisable by a nondeterministic machine using polynomial space.

Inclusions

We can show that the following inclusions hold among the complexity classes we have defined above (some of these inclusions are easier to prove than others):

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

Moreover, since the classes L, P and PSPACE are all closed under complementation, we can strengthen some of these to the following:

$$\mathsf{L} \subseteq \mathsf{NL} \cap \mathsf{co\text{-}NL}, \mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{co\text{-}NP} \quad \text{and} \quad \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \cap \mathsf{co\text{-}NPSPACE}.$$

To prove these inclusions, we show the following general inclusions hold for any constructible function f:

1. $\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$;
2. $\mathsf{TIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$;
3. $\mathsf{NTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$;
4. $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(k^{\log n + f(n)})$ for some constant k;

Of these, 1 and 2 are straightforward from the definitions, as any deterministic machine is just a nondeterministic machine in which the transition relation δ happens to be functional. On the other hand, 3 and 4 establish the relationships beteween time-bounded and space-bounded complexity classes.

The inclusion in 3 is an easy simulation. A deterministic machine can simulate a nondeterministic machine M by backtracking. In order to do this, it has to keep track of the current configuration of M as well as all the choices taken wherever a nondeterministic choice was available to M. The space required to record these choices is clearly no more than a

constant multiple of the length of the computation. Moreover, the space required to store the configuration of M also cannot be more than the number of steps in the computation of M so far. So, the total work space required by the simulating machine is at $O(f)$, where f is the time bound on M.

Inclusion 4 requires some more work to establish.

Reachability To establish the fourth of the inclusions, namely that

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(k^{\log n + f(n)})$$

we analyse the graph reachability problem. This problem is central to our understanding of nondeterministic space complexity classes. Before we begin its study, it is worth pointing out that the above mentioned inclusion implies that $\mathsf{NL} \subseteq \mathsf{P}$. This is because if we let $f(n)$ be $\log n$ in the inclusion, we have that:

$$\mathsf{NSPACE}(\log n) \subseteq \mathsf{TIME}(k^{2\log n}) = \mathsf{TIME}(n^{2\log k}).$$

The class on the right is clearly contained in P.

The Reachability problem is defined as the problem where, given as input a directed graph $G = (V, E)$, and two nodes $a, b \in V$ we are to decide whether there is a path from a to b in G. We have earlier seen an algorithm for this that requires time $O(n^2)$ and space $O(n)$. We now demonstrate a nondeterministic algorithm for Reachability that only requires logarithmic workspace, thereby establising that the problem is in NL. The algorithm is the following:

1. write the index of node a in the work space;
2. if i is the index currently written on the work space:
   (a) if $i = b$ then accept, else
   guess an index j ($\log n$ bits) and write it on the work space.
   (b) if (i, j) is not an edge, reject, else replace i by j and return to (2).

In the above description, to "guess an index j" means to perform $\log n$ steps, each of which has a nondeterministic choice of either writing a 0 or a 1 on a work tape and moving to the right. At the end of these steps, $\log n$ bits have been written. Moreover, for every index j, there is a computation path that results in j being written on the tape. Essentially, this algorithm can be seen as trying all possible indices j in parallel. For those j for which there is is an edge (i, j), the computation can continue. If there is any path from a to b in the graph, there will be a computation of this machine which successively visits all the nodes on that path.

The space requirements of the above algorithm are simple. It needs to store two indices, each of $\log n$ bits, and therefore uses $O(\log n)$ space. Hence Reachability is in NL.

The significance of the fact that Reachability is in NL is in that this problem can stand in for all problems in NL. There is a precise sense in which this is true. See Exercise sheet 3 for details. Here, we will just note that the fact that there is a polynomial time deterministic

algorithm for Reachability can be used to show that all problems in NL are in P. In general, we wish to show that for any constructible function f, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(k^{2f(n)})$.

Suppose M is a nondeterministic machine working with workspace bounded by $f(n)$ for inputs of length n. For a given input string x of length n there is a fixed finite number of configurations of M that are possible. The finite state control can be in any of q states, where q is a number which does not depend on x at all. The work tape can have one of $s^{f(n)}$ strings on it, where s is the number of distinct symbols in the tape alphabet. The head on the input tape can be in one of n different positions, and the head on the work tape can be in one of $f(n)$ different positions. Thus, the total number of distinct configurations is no more than $qnf(n)s^{f(n)}$. For some constant c, this is less than $nc^{f(n)}$.

We define the *configuration graph* of the machine M on input x to be the graph whose nodes are all possible configurations of M with x on the input tape, and the work tape having at most $f(|x|)$ symbols, and there is an edge between two configurations i and j if, and only if, $i \rightarrow_M j$, i.e. the machine M can make the transition from configuration i to configuration j in one step.

Then, it is clear that M accepts x if, and only if, there is a path from the starting configuration $(s, \triangleright, x, \triangleright, \varepsilon)$ to an accepting configuration (that is a configuration with state acc). So, the problem of determining whether M accepts x is the same as the graph reachability problem on the configuration graph of M on x. We have a deterministic algorithm that solves the graph reachability problem in time $O(n^2)$. Thus, given any nondeterministic machine M that runs using workspace $f(n)$, we can define a deterministic machine that accepts the same language which runs by first generating the configuration graph of M on the given input x, and then using the Reachability algorithm. The time taken by this deterministic machine is $O(g^2)$, where g is the size of the configuration graph. That is the time is at most $c'(nc^{f(n)})^2$ for some constant c'. But this is $k^{\log n + f(n)}$, for some constant $k \leq c'c^2$.

In addition to establishing that $\mathsf{NL} \subseteq \mathsf{P}$, this also shows that $\mathsf{NPSPACE} \subseteq \mathsf{EXP}$.

Savitch's Theorem

We can get more information about nondeterministic space complexity classes by examining other algorithms for Reachability. In particular, we can show that Reachability is solvable by a *deterministic* algorithm which uses only $O((\log n)^2)$ space. If we are only concerned about space, and not about time, this is an improvement on the deterministic algorithm we saw before. Consider the following recursive algorithm for determining if there is a path in the graph from a to b of length i or less.

```
Path(a,b,i).

 if i=1 and a =/= b and there is no edge (a,b)
    then reject
 else if there is an edge (a,b) or a=b
    then accept
 else for each vertex x
        if Path(a,x,floor(i/2)) and Path(x,b,ceil(i/2)) then accept
```

Where `ceil(i/2)` is $\lceil i/2 \rceil$ and `floor(i/2)` is $\lfloor i/2 \rfloor$.

There is a path from a to b in a graph G with n vertices if, and only if, there is a path of length n or less. So, we can solve the reachability problem by checking if $\texttt{Path}(a, b, n)$ holds. To analyse the space complexity of this algorithm, observe that the recursion can be implemented by keeping a stack of records, each of which contains a triple (a, b, i). The candidate middle vertex x (used as a loop variable) can be implemented as a counter that takes values between 1 and n, and therefore requires $\log n$ bits. Each activation record on the stack can be represented using $3 \log n$ bits ($\log n$ for each of the three components). The maximum depth of recursion is at most $\log n$, since the value of i is halved at each nested recursive call. Moreover, for each nested call, at most two activation records are placed on the stack. Thus, we need space on the stack for at most $2 \log n$ records. It follows that $6(\log n)^2$ bits of space on the stack suffice. The algorithm therefore uses space $O((\log n)^2)$.

We can use this algorithm to show that for any constructible function f such that $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$. The idea, once again, is to solve the Reachability problem on the configuration graph of a nondeterministic machine M, which uses workspace $O(f(n))$. The configuration graph has $g = c^{\log n + f(n)}$ nodes, for some constant c, and therefore the reachability problem can be solved using space

$$O((\log g)^2) = O((\log n + f(n))^2) = O((f(n))^2).$$

The last of these equalities follows from the fact that $f(n) \geq \log n$. .

This would work, except that, in order to run the $O((\log n)^2)$ algorithm for reachability, we have to first produce the configuration graph on tape. And the tape that contains the configuration graph is part of the work space of the machine that simulates M. However, the configuration graph has $c^{\log n + f(n)}$ nodes, and therefore takes more than $f(n)^2$ space. The solution is that we do not keep the entire configuration graph on tape. Rather, whenever we need to look up the graph, that is, when we need to check for a pair (i, j) of configurations whether there is an edge between them, we write out the pair of configurations and check, by looking at the machine M, whether configuration j can be reached from i in one step. In effect, the description of M serves as a compact description of the configuration graph. With this, we can see that the total amount of work space needed is no more than $O((f(n))^2)$.

From the inclusion $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$ follows Savitch's theorem:

Theorem

$\mathsf{PSPACE} = \mathsf{NPSPACE}$.

From which it also follows that $\mathsf{NPSPACE} = \mathsf{co\text{-}NPSPACE}$, since PSPACE is closed under complementation. However, a more general result about the closure of nondeterministic space classes under complementation is known. Immerman and Szelepcśenyi proved that for any constructible function f with $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) = \mathsf{co\text{-}NSPACE}(f(n))$. The proof is based on a still more clever algorithm for Reachability, which shows that there is a nondeterministic machine, which with $O(\log n)$ work space determines the number of nodes reachable from a node a.

Provable Intractability

Often, when we show that a problem is NP-complete, it is said that we have proved the problem intractable. Of course, this is not, strictly speaking, correct. A proof of NP-completeness does not show that the given problem is not in P unless we can also show that $\mathsf{P} \neq \mathsf{NP}$. Our aim in this section is to show that there are some problems for which it is possible to prove that they are not in P categorically, i.e. without the assumption that $\mathsf{P} \neq \mathsf{NP}$.

Hierarchy

While proving lower bounds for specific problems such as the travelling salesman problem remains the holy grail of complexity theory, one instance where we know how to prove lower bounds is for problems that are constructed specifically for this purpose. That is, we can use diagonalisation to construct a language with a specific lower bound. This allows us to show, in particular that increasing the time (or space) bounds on a complexity class does give us the ability to recognise more languages.

One such hierarchy theorem we can show is:

Time Hierarchy Theorem

For any constructible function f with $f(n) \geq n$, $\mathsf{TIME}(f(n))$ is properly contained in $\mathsf{TIME}(f(2n+1)^2)$.

To see this, we define a version of the halting problem with time bound f. That is, define the language:

$$H_f = \{[M], x \mid M \text{ accepts } x \text{ in } f(|x|) \text{ steps}\}$$

We now make two observations. First:

$$H_f \in \mathsf{TIME}(f(n)^2).$$

A machine for recognising H_f would first compute $f(|x|)$, and on a separate work tape, write out 0, $f(|x|)$ times, to use as a clock for the rest of the computation. It would then simulate machine M on input x for $f(|x|)$ many steps, at each step looking through the description of M given for the appropriate transition. The calculation of the time bound is left as an exercise.

The second observation is:

$$H_f \notin \mathsf{TIME}(f(\lfloor n/2 \rfloor)).$$

The argument for this is similar to the argument that the halting problem H is undecidable. Suppose $H_f \in \mathsf{TIME}(f(\lfloor n/2 \rfloor))$. Then, we can construct a machine N which accepts $[M]$ if, and only if, $[M], [M] \notin H_f$. The machine simply copies $[M]$, inserting a comma between the two copies, and then runs the machine that accepts H_f. Moreover, the running time of N on an input of length n is $f(\lfloor (2n+1)/2 \rfloor) = f(n)$. We can now ask whether N accepts the input $[N]$, and we see that we get a contradiction either way.

From these two observations, the Time Hierarchy Theorem immediately follows.

Among the consequences of the Time Hierarchy Theorem is that there is no fixed k such that all languages in P can be decided in time $O(n^k)$. Another consequence is that the complexity class EXP, defined by:

$$\mathsf{EXP} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(2^{n^k}),$$

is a proper extension of P. That is, $\mathsf{P} \subseteq \mathsf{EXP}$, but $\mathsf{EXP} \not\subseteq \mathsf{P}$.

Similar results can be obtained for space complexity, by proving a Space Hierarchy Theorem. See Exercise Sheet 4.