Complexity Theory
Lectures 1–6

Lecturer: Dr. Timothy G. Griffin

Slides by Anuj Dawar

Computer Laboratory
University of Cambridge
Easter Term 2009

http://www.cl.cam.ac.uk/teaching/0809/Complexity/

Texts

The main texts for the course are:

Computational Complexity.
Christos H. Papadimitriou.

Introduction to the Theory of Computation.
Michael Sipser.

Other useful references include:

Computers and Intractability: A guide to the theory of NP-completeness.
Michael R. Garey and David S. Johnson.

Structural Complexity. Vols I and II.
J.L. Balcázar, J. Díaz and J. Gabarró.

Computability and Complexity from a Programming Perspective.
Neil Jones.

Outline

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

Outline - contd.

- **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** 5.6, 5.7.

Complexity Theory

Complexity Theory seeks to understand what makes certain problems algorithmically difficult to solve.

In Data Structures and Algorithms, we saw how to measure the complexity of specific algorithms, by asymptotic measures of number of steps.

In Computation Theory, we saw that certain problems were not solvable at all, algorithmically.

Both of these are prerequisites for the present course.

Algorithms and Problems

> Insertion Sort runs in time $O(n^2)$, while Merge Sort is an $O(n \log n)$ algorithm.

The first half of this statement is short for:

> 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.

It makes sense to compare the two algorithms, because they seek to solve the same problem.

But, what is the complexity of the sorting problem?

Lower and Upper Bounds

What is the running time complexity of the fastest algorithm that sorts a list?

By the analysis of the Merge Sort algorithm, we know that this is no worse than $O(n \log n)$.

The complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem.

To establish a *lower bound*, we need to show that no possible algorithm, including those as yet undreamed of, can do better.

In the case of sorting, we can establish a lower bound of $\Omega(n \log n)$, showing that Merge Sort is asymptotically optimal.

Sorting is a rare example where known upper and lower bounds match.

Review

The complexity of an algorithm (whether measuring number of steps, or amount of memory) is usually described asymptotically:

Definition

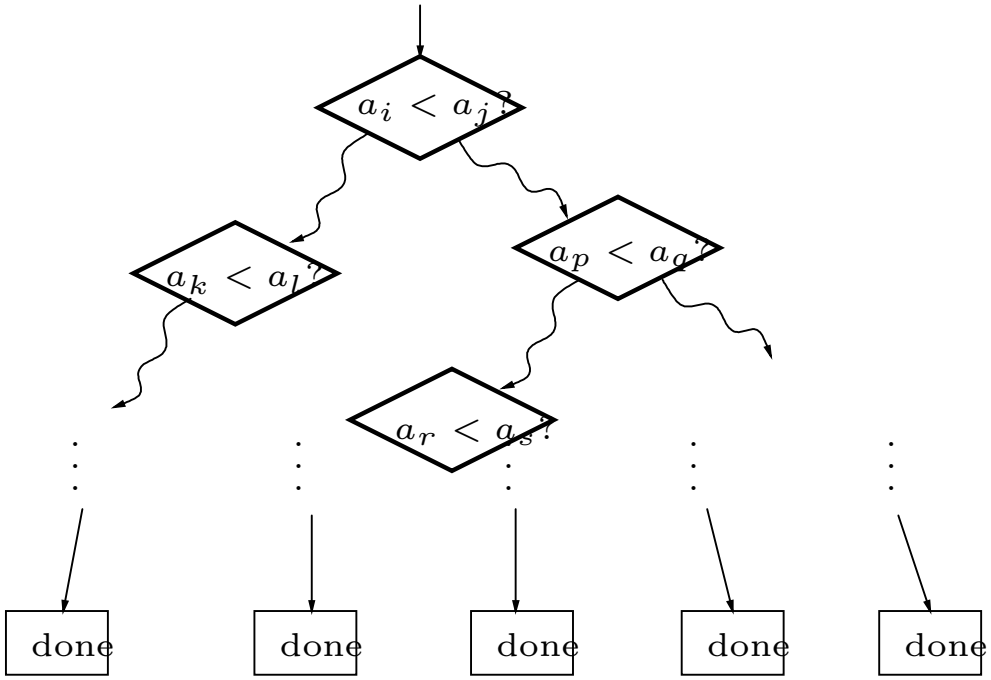
For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, 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 $f = \Omega(g)$.

Usually, O is used for upper bounds and Ω for lower bounds.

Lower Bound on Sorting

An algorithm A sorting a list of n distinct numbers $a_1, \ldots, a_n$.



To work for all permutations of the input list, the tree must have at least $n!$ leaves and therefore height at least $\log_2(n!) = \theta(n \log n)$.

Travelling Salesman

Given

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

Complexity of TSP

Obvious algorithm: Try all possible orderings of V and find the one with lowest cost.
The worst case running time is $\theta(n!)$.

Lower bound: An analysis like that for sorting shows a lower bound of $\Omega(n \log n)$.

Upper bound: The currently fastest known algorithm has a running time of $O(n^2 2^n)$.

Between these two is the chasm of our ignorance.

Formalising Algorithms

To prove a lower bound on the complexity of a problem, rather than a specific algorithm, we need to prove a statement about all algorithms for solving it.

In order to prove facts about all algorithms, we need a mathematically precise definition of algorithm.

We will use the *Turing machine*.

> The simplicity of the Turing machine means it's not useful for actually expressing algorithms, but very well suited for proofs about all algorithms.

Turing Machines

For our purposes, a Turing Machine consists of:

- K — a finite set of states;
- Σ — a finite set of symbols, including $\sqcup$.
- $s \in K$ — an initial state;
- $\delta : (K \times \Sigma) \to (K \cup \{a, r\}) \times \Sigma \times \{L, R, S\}$

  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)

Configurations

A complete description of the configuration of a machine can be given if we know what state it is in, what are the contents of its tape, and what is the position of its head. This can be summed up in a simple triple:

Definition

A *configuration* is a triple (q, w, u), where $q \in K$ 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 completely determines the future behaviour of the machine.

Computations

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

$$(q, w, u) \rightarrow_M (q', w', u')$$

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.

Computations

The relation $\rightarrow_M^\star$ is the reflexive and transitive closure of $\rightarrow_M$.

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

The language $L(M) \subseteq \Sigma^\star$ *accepted* by the machine M is the set of strings

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

A machine M is said to *halt on input* x if for some w and u, either $(s, \triangleright, x) \rightarrow_M^\star (\text{acc}, w, u)$ or $(s, \triangleright, x) \rightarrow_M^\star (\text{rej}, w, u)$

Decidability

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 language L is *semi-decidable* if it is recursively enumerable.

A function $f : \Sigma^\star \to \Sigma^\star$ is *computable*, if there is a machine M, such that for all x, $(s, \triangleright, x) \to^\star_M (\text{acc}, f(x), \varepsilon)$

Example

Consider the machine with δ given by:

	$\triangleright$	0	1	$\sqcup$
s	$s, \triangleright, R$	$s, 0, R$	$s, 1, R$	$q, \sqcup, L$
q	$\text{acc}, \triangleright, R$	$q, \sqcup, L$	$\text{rej}, \sqcup, R$	$q, \sqcup, L$

This machine will accept any string that contains only 0s before the first blank (but only after replacing them all by blanks).

Multi-Tape Machines

The formalisation of Turing machines extends in a natural way to multi-tape machines. For instance a machine with k tapes is specified by:

- K, Σ, s; and
- $\delta : (K \times \Sigma^k) \to K \cup \{a, r\} \times (\Sigma \times \{L, R, S\})^k$

Similarly, a configuration is of the form:

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

Complexity

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 $M = (K, \Sigma, s, \delta)$, such that:

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

Similarly, we define $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses $O(f(n))$ tape cells on inputs of length n.

In defining space complexity, we assume a machine M, which has a read-only input tape, and a separate work tape. We only count cells on the work tape towards the complexity.

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*.

$$\delta \subseteq (K \times \Sigma) \times (K \cup \{a, r\} \times \Sigma \times \{R, L, S\}).$$

The yields relation $\rightarrow_M$ is also no longer functional.

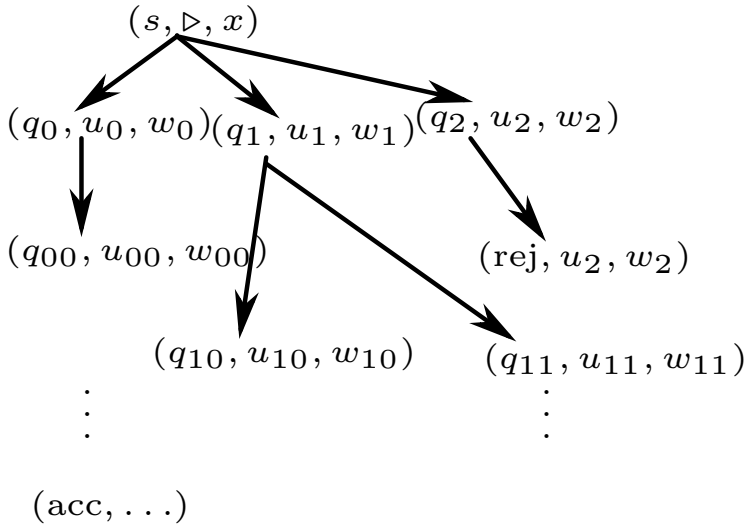
We still define the language accepted by M by:

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

though, for some x, there may be computations leading to accepting as well as rejecting states.

Computation Trees

With a nondeterministic machine, each configuration gives rise to a tree of successive configurations.



Decidability and Complexity

For every decidable language L, there is a computable function f such that

$$L \in \mathsf{TIME}(f(n))$$

If L is a semi-decidable (but not decidable) language accepted by M, then there is no computable function f such that every accepting computation of M, on input of length n is of length at most $f(n)$.

Complexity Classes

A complexity class is a collection of languages determined by three things:

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

Polynomial Bounds

By making the bounds broad enough, we can make our definitions fairly independent of the model of computation.

> The collection of languages recognised in *polynomial time* is the same whether we consider Turing machines, register machines, or any other deterministic model of computation.
>
> The collection of languages recognised in *linear time*, on the other hand, is different on a one-tape and a two-tape Turing machine.

We can say that being recognisable in polynomial time is a property of the language, while being recognisable in linear time is sensitive to the model of computation.

Polynomial Time

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

The class of languages decidable in polynomial time.

The complexity class P plays an important role in our theory.

- It is robust, as explained.
- It serves as our formal definition of what is *feasibly computable*

One could argue whether an algorithm running in time $\theta(n^{100})$ is feasible, but it will eventually run faster than one that takes time $\theta(2^n)$.

Making the distinction between polynomial and exponential results in a useful and elegant theory.

Example: Reachability

The Reachability decision problem is, given a *directed* graph $G = (V, E)$ and two nodes $a, b \in V$, to determine whether there is a path from a to b in G.

A simple search algorithm as follows solves it:

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.

Analysis

This algorithm requires $O(n^2)$ time and $O(n)$ space.

The description of the algorithm would have to be refined for an implementation on a Turing machine, but it is easy enough to show that:

$$\text{Reachability} \in \mathsf{P}$$

To formally define Reachability as a language, we would have to also choose a way of representing the input (V, E, a, b) as a string.

Example: Euclid's Algorithm

Consider the decision problem (or *language*) RelPrime defined by:

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

The standard algorithm for solving it is due to Euclid:

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.

Analysis

The number of repetitions at step 2 of the algorithm is at most $O(\log x)$.

why?

This implies that RelPrime is in P.

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.

Primality

Consider the decision problem (or *language*) Prime defined by:

$$\{x \mid x \text{ is prime}\}$$

The obvious algorithm:

For all y with $1 < y \leq \sqrt{x}$ check whether $y|x$.

requires $\Omega(\sqrt{x})$ steps and is therefore *not* polynomial in the length of the input.

Is Prime $\in$ P?

Boolean Expressions

Boolean expressions are built up from an infinite set of variables

$$X = \{x_1, x_2, \ldots\}$$

and the two constants `true` and `false` by the 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 $(\phi \vee \psi)$.

Evaluation

If an expression contains no variables, then it can be evaluated to either `true` or `false`.

Otherwise, it can be evaluated, *given* a truth assignment to its variables.

Examples:

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

$(x_1 \vee \texttt{false}) \wedge ((\neg x_1) \vee x_2)$

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

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

Boolean Evaluation

There is a deterministic Turing machine, which given a Boolean expression *without variables* of length n will determine, in time $O(n^2)$ whether the expression evaluates to `true`.

The algorithm works by scanning the input, rewriting formulas according to the following rules:

Rules

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

Analysis

Each scan of the input ($O(n)$ steps) must find at least one subexpression matching one of the rule patterns.

Applying a rule always eliminates at least one symbol from the formula.

Thus, there are at most $O(n)$ scans required.

The algorithm works in $O(n^2)$ steps.

Satisfiability

For Boolean expressions ϕ that contain variables, we can ask

> Is there an assignment of truth values to the variables which would make the formula evaluate to `true`?

The set of Boolean expressions for which this is true is the language SAT of *satisfiable* expressions.

This can be decided by a deterministic Turing machine in time $O(n^2 2^n)$.

An expression of length n can contain at most n variables.

For each of the 2^n possible truth assignments to these variables, we check whether it results in a Boolean expression that evaluates to `true`.

Is $\mathsf{SAT} \in \mathsf{P}$?

Circuits

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 $l(v) \in \{\vee, \wedge\}$

The value of the expression is given by the value at node n.

CVP

A circuit is a more compact way of representing a Boolean expression.

Identical subexpressions need not be repeated.

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.

Composites

Consider the decision problem (or *language*) Composite defined by:

$$\{x \mid x \text{ is not prime}\}$$

This is the complement of the language Prime.

Is Composite $\in$ P?

Clearly, the answer is yes if, and only if, Prime $\in$ P.

Hamiltonian Graphs

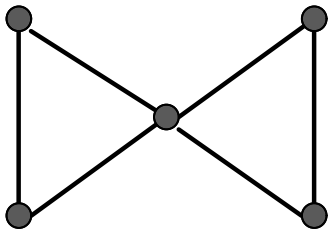
Given a graph $G = (V, E)$, a *Hamiltonian cycle* in G is a path in the graph, starting and ending at the same node, such that every node in V appears on the cycle *exactly once*.

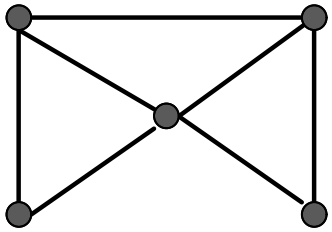
A graph is called *Hamiltonian* if it contains a Hamiltonian cycle.

The language HAM is the set of encodings of Hamiltonian graphs.

Is HAM $\in$ P?

Examples





The first of these graphs is not Hamiltonian, but the second one is.

Polynomial Verification

The problems Composite, SAT and HAM have something in common.

In each case, there is a *search space* of possible solutions.

> the factors of x; a truth assignment to the variables of ϕ; a list of the vertices of G.

The number of possible solutions is *exponential* in the length of the input.

Given a potential solution, it is *easy* to check whether or not it is a solution.

Verifiers

A verifier V for a language L is an 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*.

Many natural examples arise, whenever we have to construct a solution to some design constraints or specifications.

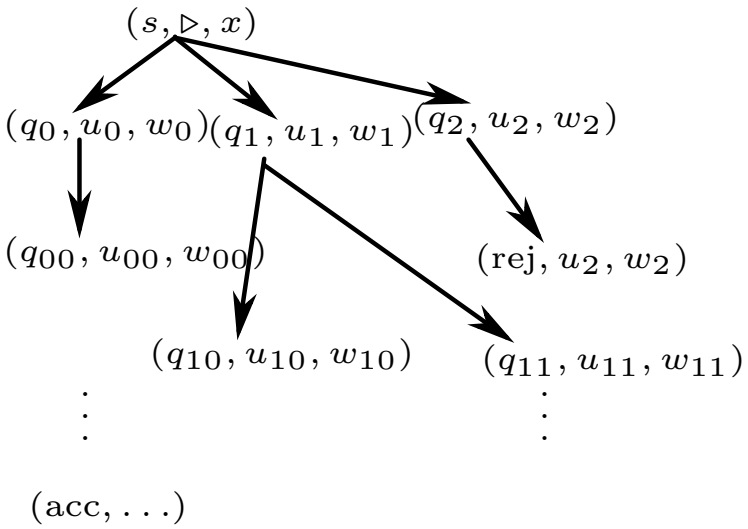
Nondeterministic Complexity Classes

We have already defined $\mathsf{TIME}(f(n))$ and $\mathsf{SPACE}(f(n))$.

$\mathsf{NTIME}(f(n))$ is defined as the class of those languages L which are accepted by a *nondeterministic* Turing machine M, such that for every $x \in L$, there is an accepting computation of M on x of length at most $f(n)$.

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

Nondeterminism



For a language in NTIME($f(n)$), the height of the tree is bounded by $f(n)$ when the input is of length n.

NP

A language L is polynomially verifiable if, and only if, it is in NP.

To prove this, suppose L is a language, which has a verifier V, which runs in time $p(n)$.

The following describes 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)

NP

In the other direction, suppose M is a nondeterministic machine that accepts a language L in time n^k.

We define the *deterministic algorithm* V which on input (x, c) simulates M on input x.

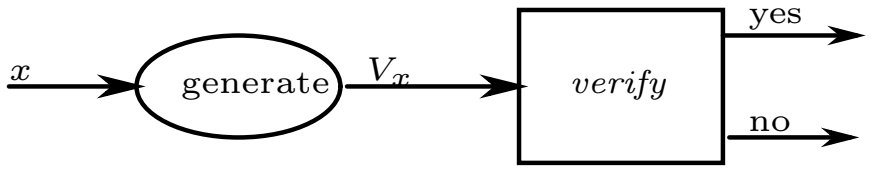
At the i^{th} nondeterministic choice point, V looks at the i^{th} character in c to decide which branch to follow.

If M accepts then V accepts, otherwise it rejects.

V is a polynomial verifier for L.

Generate and Test

We can think of nondeterministic algorithms in the generate-and test paradigm:



Where the *generate* component is nondeterministic and the *verify* component is deterministic.

Reductions

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 $x \in \Sigma_1^\star$,

$$f(x) \in L_2 \text{ if, and only if, } x \in L_1$$

Resource Bounded Reductions

If f is computable by a polynomial time algorithm, we say that L_1 is *polynomial time reducible* to L_2.

$$L_1 \leq_P L_2$$

If f is also computable in $\mathsf{SPACE}(\log n)$, we write

$$L_1 \leq_L L_2$$

Reductions 2

If $L_1 \leq_P L_2$ we understand that L_1 is no more difficult to solve than L_2, at least as far as polynomial time computation is concerned.

That is to say,

> If $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$, then $L_1 \in \mathsf{P}$

We can get an algorithm to decide L_1 by first computing f, and then using the polynomial time algorithm for L_2.

Completeness

The usefulness of reductions is that they allow us to establish the *relative* complexity of problems, even when we cannot prove absolute lower bounds.

Cook (1972) first showed that there are problems in NP that are maximally difficult.

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.

SAT is NP-complete

Cook showed that the language SAT of satisfiable Boolean expressions is NP-complete.

To establish this, 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, there is a nondeterministic Turing machine

$$M = (K, \Sigma, s, \delta)$$

and a bound n^k such that a string x is in L if, and only if, it is accepted by M within n^k steps.

Boolean Formula

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 input x.

$f(x)$ has the following variables:

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

Intuitively, these variables are intended to mean:

- $S_{i,q}$ – the state of the machine at time i is q.
- $T_{i,j,\sigma}$ – at time i, the symbol at position j of the tape is σ.
- $H_{i,j}$ – at time i, the tape head is pointing at tape cell j.

We now have to see how to write the formula $f(x)$, so that it enforces these meanings.

Initial state is s and the head is initially at the beginning of the tape.

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

The head is never in two places at once

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

The machine is never in two states at once

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

Each tape cell contains only one symbol

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

The initial tape contents are x

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

The tape does not change except under the head

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

Each step is according to δ.

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

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

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

Finally, some accepting state is reached

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

CNF

A Boolean expression is in *conjunctive normal form* 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.

For any Boolean expression ϕ, there is an equivalent expression ψ in conjunctive normal form.

ψ can be exponentially longer than ϕ.

However, CNF-SAT, the collection of satisfiable CNF expressions, is NP-complete.

3SAT

A Boolean expression is in 3CNF if it is in conjunctive normal form and each clause contains at most 3 literals.

3SAT is defined as the language consisting of those expressions in 3CNF that are satisfiable.

3SAT is NP-complete, as there is a polynomial time reduction from CNF-SAT to 3SAT.

Composing Reductions

Polynomial time reductions are clearly closed under composition.

So, if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then we also have $L_1 \leq_P L_3$.

Note, this is also true of $\leq_L$, though less obvious.

If we show, for some problem A in NP that

$$\mathsf{SAT} \leq_P A$$

or

$$\mathsf{3SAT} \leq_P A$$

it follows that A is also NP-complete.

Independent Set

Given a graph $G = (V, E)$, a subset $X \subseteq V$ of the vertices is said to be an *independent set*, if there are no edges (u, v) for $u, v \in X$.

The natural algorithmic problem is, given a graph, find the largest independent set.

To turn this *optimisation problem* into a *decision problem*, we define IND 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.

IND is clearly in NP. We now show it is NP-complete.

Reduction

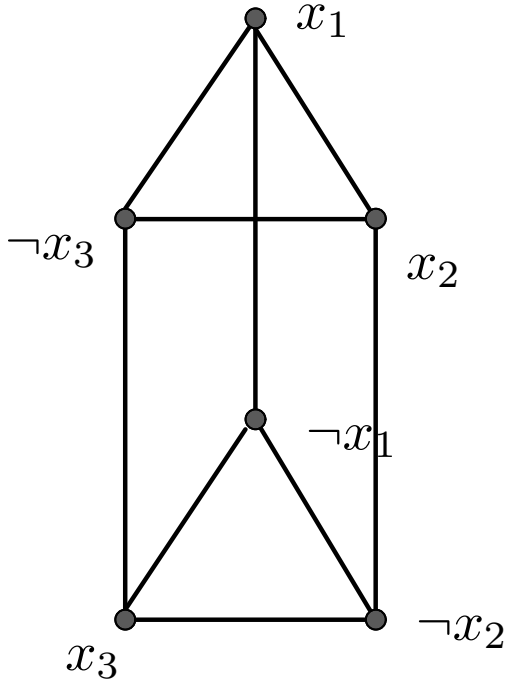
We can construct a reduction from 3SAT to IND.

A Boolean expression ϕ in 3CNF with m clauses is mapped by the reduction to the pair (G, m), where G is the graph obtained from ϕ as follows:

> G contains m triangles, one for each clause of ϕ, with each node representing one of the literals in the clause.
>
> Additionally, there is an edge between two nodes in different triangles if they represent literals where one is the negation of the other.

Example

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



Clique

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.

As with IND, we can define a decision problem version:

CLIQUE is defined as:

> 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.

Clique 2

CLIQUE is in NP by the algorithm which *guesses* a clique and then verifies it.

CLIQUE is NP-complete, since

IND $\leq_P$ CLIQUE

by the reduction that maps the pair (G, K) to $(\bar{G}, K)$, where $\bar{G}$ is the complement graph of G.

k-Colourability

A graph $G = (V, E)$ 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 gives rise to a decision problem for each k.

2-colourability is in P.

For all $k > 2$, k-colourability is NP-complete.

3-Colourability

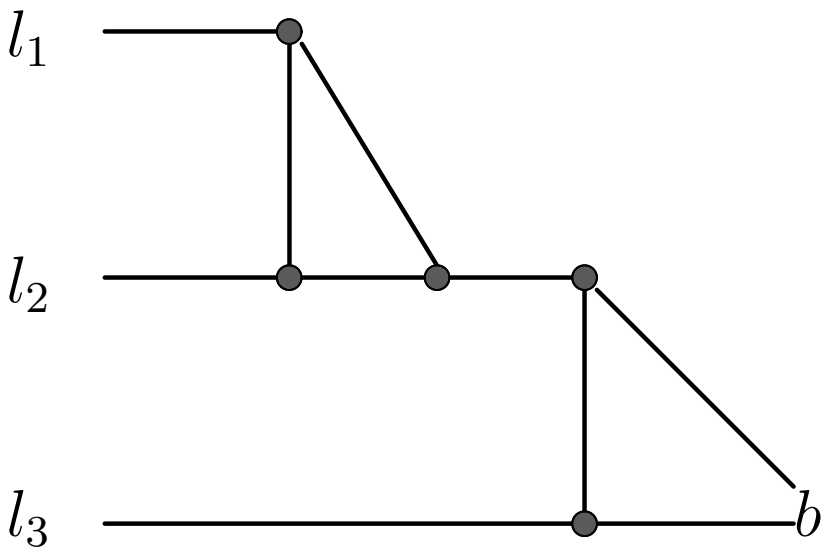
3-Colourability is in NP, as we can *guess* a colouring and verify it.

To show NP-completeness, we can construct a reduction from 3SAT to 3-Colourability.

For each variable x, have two vertices x, $\bar{x}$ which are connected in a triangle with the vertex a (common to all variables).

In addition, for each clause containing the literals l_1, l_2 and l_3 we have a gadget.

Gadget



With a further edge from a to b.