Complexity Theory

Anuj Dawar

Computer Laboratory
University of Cambridge
Easter Term 2001

Texts

The main text for the course is:

Computational Complexity.
Christos H. Papadimitriou.

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.

- **Algorithms and problems.** 1.1–1.3.
- **Time and space.** 2.1–2.5, 2.7.
- **Complexity classes. Hierarchy.** 7.1–7.2.
- **Reachability.** 7.3
- **Boolean logic.** 4.1–4.3
- **NP-completeness.** 8.1–8.2, 9.1–9.2.
- **Graph-theoretic problems.** 9.3
- **Sets, numbers and scheduling.** 9.4
- **coNP.** 10.1–10.2.
- **Cryptographic complexity.** 12.1–12.2.

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

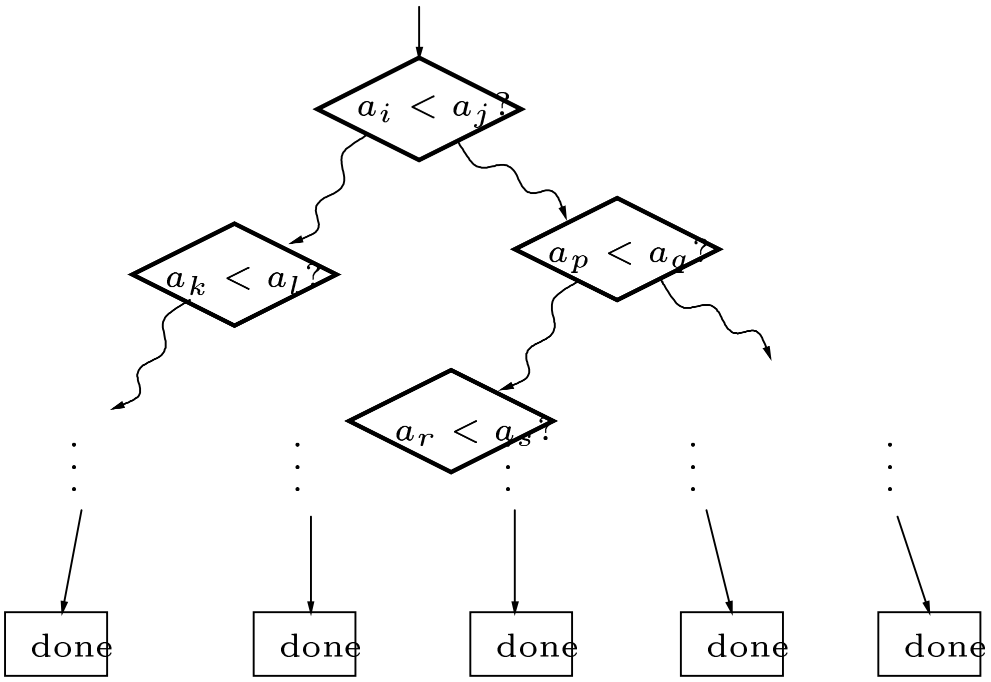
For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, 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 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.

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.

Turing Machines

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

For our purposes, a Turing Machine consists of:

- K — a finite set of states;
- Σ — a finite set of symbols, disjoint from K;
- $s \in K$ — 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 $u = cx$.

Computations

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

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

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

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

A machine M is said to *halt on input* x if the set of configurations (q, w, u) such that
$(s, \triangleright, x) \to_M^\star (q, w, u)$
is finite.

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 but not decidable.

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

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	$\mathrm{acc}, \triangleright, R$	$q, \sqcup, L$	$\mathrm{rej}, \sqcup, R$	$q, \sqcup, L$

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
- for each $x \in L$ with n symbols, there is a computation of M, of length at most $f(n)$ starting with $(s, \triangleright, x)$ and ending in an accepting configuration.

Similarly, we define $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses at most $f(n)$ 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.

Non-Determinism

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 *non-deterministic Turing machine*.

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

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

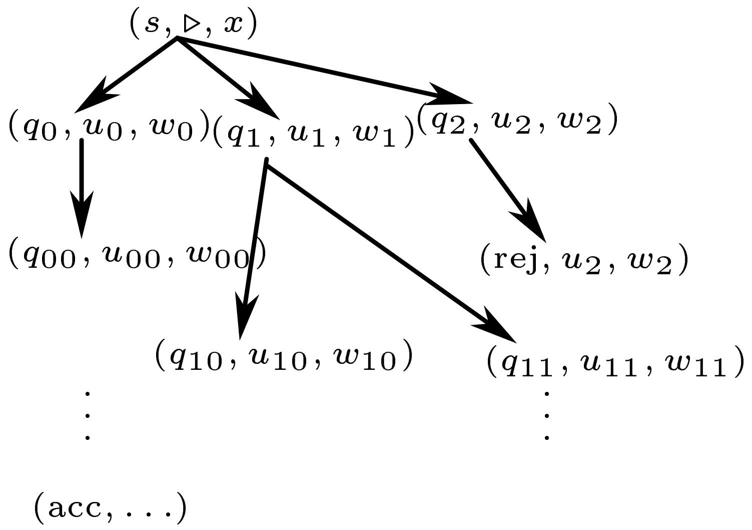
We still define the language accepted by M by:

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

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

Computation Trees

With a non-deterministic 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 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 non-deterministic 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.

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

Constructible Functions

A complexity class such as $\mathsf{TIME}(f(n))$ can be very unnatural, if $f(n)$ is.

From now on, we restrict our bounding functions $f(n)$ to be proper 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 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.*

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 $f(g)$ (this last, provided that $f(n) > n$).

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 *non-deterministic* Turing machine M, such that for every $x \in L$, there is an accepting computation of M on x of length at most $f(n)$.

$\mathsf{NSPACE}(f(n))$ is the class of languages accepted by a *non-deterministic* Turing machine using at most $f(n)$ work space.

If $f(n)$ is constructible, we can always choose M so that it always halts (accepting or rejecting) using only $f(n)$ time (or space, as the case may be).

Classes

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

The class of languages decidable in polynomial time.

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

$\mathsf{L} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(k \cdot \log n)$

$\mathsf{NL} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(k \cdot \log n)$

$\mathsf{PSPACE} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k)$

The class of languages decidable in polynomial space.

$\mathsf{NPSPACE} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k)$

Complement Classes

Also, define

co-NL – the languages whose complements are in NL.

co-NP – the languages whose complements are in NP.

co-NPSPACE – the languages whose complements are in NPSPACE.

Complexity classes defined in terms of non-deterministic machine models are not necessarily closed under complementation of languages.

Inclusions

We have the following inclusions:

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

Moreover,

$$\mathsf{L} \subseteq \mathsf{NL} \cap \mathsf{co\text{-}NL}$$

$$\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{co\text{-}NP}$$

$$\mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \cap \mathsf{co\text{-}NPSPACE}$$

Hierarchy Theorems

For any constructible function f, with $f(n) \geq n$, define the f-bounded *halting language* to be:

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

where $[M]$ is a description of M in some fixed encoding scheme.

Then, we can show

$H_f \in \mathsf{TIME}(f(n)^3)$ and $H_f \notin \mathsf{TIME}(f(\lfloor n/2 \rfloor))$

Time Hierarchy Theorem

For any constructible function $f(n) \geq n$, $\mathsf{TIME}(f(n))$ is properly contained in $\mathsf{TIME}(f(2n+1)^3)$.

Establishing Inclusions

To establish the known inclusions between the main complexity classes, we prove the following.

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

The first two are straightforward from definitions.

The third is an easy simulation.

The last requires some more work.

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 Reachability is in P.

In general, any polynomial time algorithm (on any other model of computation) is still polynomial time on a Turing machine, though the specific polynomial bound may change.

NL Reachability

We can construct an algorithm to show that the Reachability problem is in NL:

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).

We can use the $O(n^2)$ algorithm for Reachability to show that:

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

for some constant k.

Let M be a non-deterministic machine working in space bounds $f(n)$.

For any input x of length n, there is a constant c (depending on the number of states and alphabet of M) such that the total number of possible configurations of M within space bounds $f(n)$ is bounded by $n \cdot c^{f(n)}$.

> Here, $c^{f(n)}$ represents the number of different possible contents of the work space, and n different head positions on the input.

Configuration Graph

Define the *configuration graph* of M, x to be the graph whose nodes are the possible configurations, and there is an edge from i to j if, and only if, $i \rightarrow_M j$.

Then, M accepts x if, and only if, some accepting configuration is reachable from the starting configuration $(s, \triangleright, x, \varepsilon, \varepsilon)$ in the configuration graph of M, x.

Using the $O(n^2)$ algorithm for Reachability, we get that M can be simulated by a deterministic machine operating in time

$$c'(nc^{f(n)})^2 = c'c^{2(\log n + f(n))} = k^{(\log n + f(n))}$$

In particular, this establishes that $\mathsf{NL} \subseteq \mathsf{P}$ and $\mathsf{PSPACE} \subseteq \mathsf{EXP}$.

Savitch's Theorem

Further simulation results for nondeterministic space are obtained by other algorithms for Reachability.

We can show that Reachability can be solved by a *deterministic* algorithm in $O((\log n)^2)$ space.

Consider the following recursive algorithm for determining whether there is a path from a to b of length at most n (for n a power of 2):

$O((\log n)^2)$ space Reachability algorithm:

if $a = b$ then accept else, for each node x, check:

1. is there a path $a - x$ of length $n/2$; and
2. is there a path $x - b$ of length $n/2$?

if such an x is found, then accept, else reject.

The maximum depth of recursion is $\log n$, and the number of bits of information kept at each stage is $3 \log n$.

Savitch's Theorem - 2

The space efficient algorithm for reachability used on the configuration graph of a non-deterministic machine shows:

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$$

for $f(n) \geq \log n$.

This yields

$$\mathsf{PSPACE} = \mathsf{NPSPACE} = \text{co-}\mathsf{NPSPACE}$$

.

Complementation

A still more clever algorithm for Reachability has been used to show that non-deterministic space classes are closed under complementation:

If $f(n) \geq \log n$, then

$$\mathsf{NSPACE}(f(n)) = \mathsf{co\text{-}NSPACE}(f(n))$$

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:

$(\mathtt{true} \lor \mathtt{false}) \land (\neg \mathtt{false})$

$(x_1 \lor \mathtt{false}) \land ((\neg x_1) \lor x_2)$

$(x_1 \lor \mathtt{false}) \land (\neg x_1)$

$(x_1 \lor (\neg x_1)) \land \mathtt{true}$

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 be scanning the input, rewriting formulas according to the following rules:

Rules

- $(\mathtt{true} \lor \phi) \Rightarrow \mathtt{true}$
- $(\phi \lor \mathtt{true}) \Rightarrow \mathtt{true}$
- $(\mathtt{false} \lor \mathtt{false}) \Rightarrow \mathtt{false}$
- $(\mathtt{false} \land \phi) \Rightarrow \mathtt{false}$
- $(\phi \land \mathtt{false}) \Rightarrow \mathtt{false}$
- $(\mathtt{true} \land \mathtt{true}) \Rightarrow \mathtt{true}$
- $(\neg\mathtt{true}) \Rightarrow \mathtt{false}$
- $(\neg\mathtt{false}) \Rightarrow \mathtt{true}$

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 that contain variables, we can ask a different question:

> 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`.

Validity

We also define VAL—the set of *valid* expressions—to be those Boolean expressions for which every assignment of truth values to variables yields an expression equivalent to `true`.

By an algorithm similar to SAT, we see VAL is in $\mathsf{TIME}(n^2 2^n)$.

Neither SAT nor VAL is known to be in P.

Nondeterminism

There is a non-deterministic machine that will accept SAT in time $O(n^2)$.

The algorithm *guesses* a truth assignment for the variables ($O(n)$ non-deterministic steps), and then uses the deterministic $O(n^2)$ algorithm to check that this assignment satisfies the given expression.

Thus, SAT is in NP.

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.

However, we can show VAL is in co-NP, by constructing a nondeterministic, $O(n^2)$ machine which can determine whether a given Boolean expression has a *falsifying* truth assignment.

NP problems

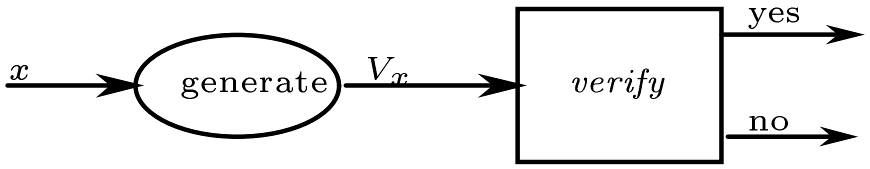
SAT is paradigmatic of NP problems, in the sense that every language L in NP can be characterised by a *search space*.

For every candidate string x, there is a (potentially exponential) search space of solutions, each of whose lengths is bounded by a polynomial in the length of x.

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

Generate and Test

Another view of non-deterministic algorithms is the generate-and test paradigm:



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

Circuits

A circuit is a 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 $E(i, j)$, 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 $l(v) \in \{\texttt{true}, \texttt{false}\}$

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

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

CVP

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.

CVP is complete for P under L reductions.

That is, for every language A in P,

$$A \leq_L \mathsf{CVP}$$

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

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

3SAT

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.

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.

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

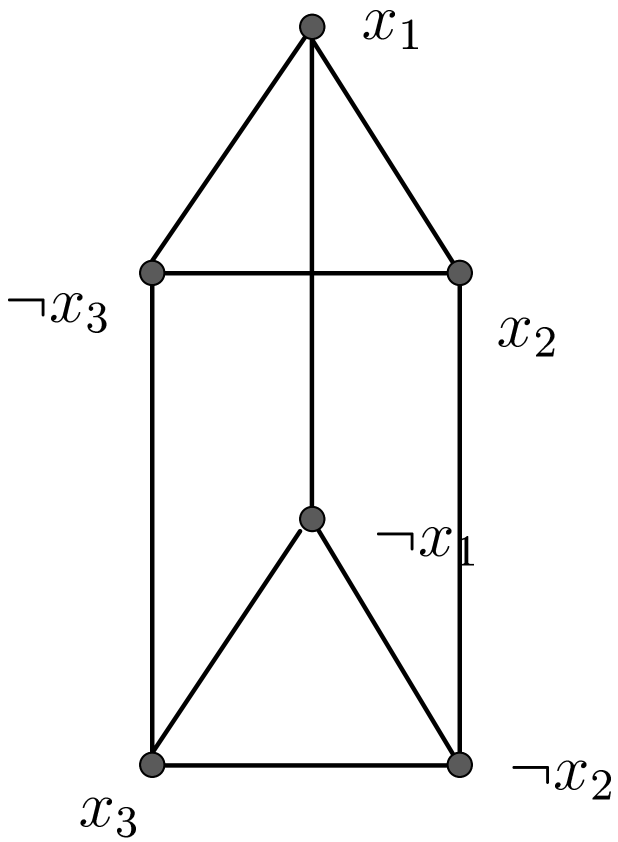
We can construct a reduction from 3SAT to IND.

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.

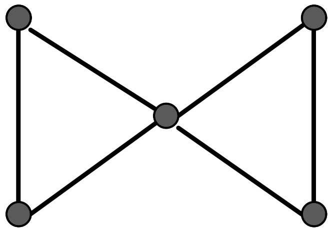
Hamiltonian Graphs

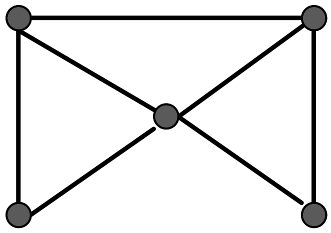
Given a graph $G = (V, E)$, a *Hamiltonian cycle* in G is a path in the graph, starting and ending at the same node, such that every node in V appears on the cycle *exactly once*.

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

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

Examples





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

Reduction

We can construct a reduction from 3SAT to HAM

Essentially, this involves coding up a Boolean expression as a graph, so that every satisfying truth assignment to the expression corresponds to a Hamiltonian circuit of the graph.

This reduction is much more intricate than the one for IND.

Travelling Salesman

As with other optimisation problems, we can make a decision problem version of the Travelling Salesman problem.

The problem TSP consists of the set of triples

$$(V, c : V \times V \to \mathbb{N}, t)$$

such that there is a tour of the set of vertices V, which under the cost matrix c, has cost t or less.

Reduction

There is a simple reduction from HAM to TSP, mapping a graph (V, E) to the triple $(V, c : V \times V \rightarrow \mathbb{N}, n)$, where

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

and n is the size of V.

Sets, Numbers and Scheduling

It is not just problems about formulas and graphs that turn out to be NP-complete.

Literally hundreds 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.

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.

We now 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.

3D Matching

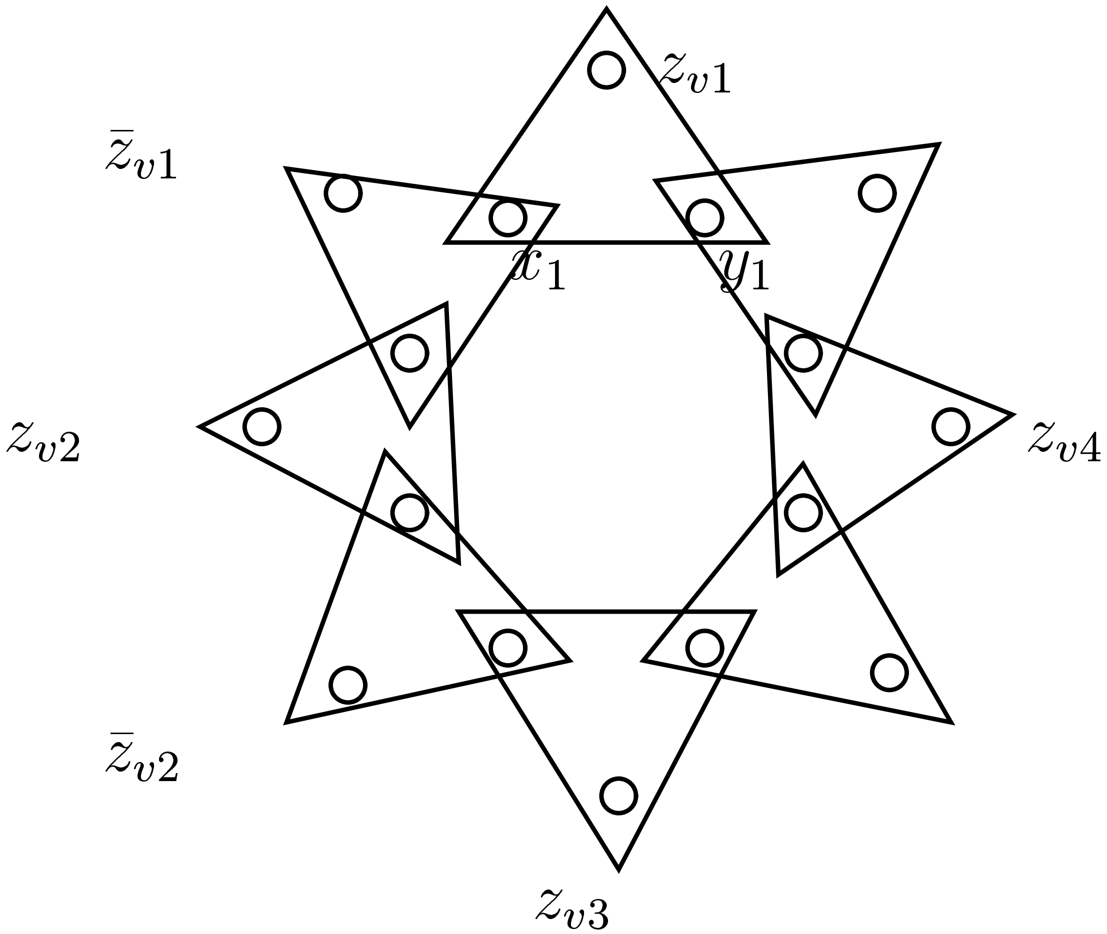
The decision problem of *3D Matching* is defined as:

> 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'?

We can show that 3DM is NP-complete by a reduction from 3SAT.

Reduction

If a Boolean expression ϕ in 3CNF has n variables, and m clauses, we construct for each variable v the following gadget.



In addition, for every clause c, we have two elements x_c and y_c.

If the literal v occurs in c, we include the triple

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

in M.

Similarly, if $\bar{v}$ occurs in c, we include the triple

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

in M.

Finally, we include extra dummy elements in X and Y to make the numbers match up.

Exact Set Covering

Two other well known problems are proved NP-complete by immediate reduction from 3DM.

Exact Cover by 3-Sets 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 reduction from 3DM simply takes $U = X \cup Y \cup Z$, and S to be the collection of three-element subsets resulting from M.

Set Covering

More generally, we have the *Set Covering* problem:

> 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?

Knapsack

KNAPSACK is a problem which generalises many natural scheduling and optimisation problems, and through reductions has been used to show many such problems NP-complete.

In the problem, 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?

Reduction

The proof that KNAPSACK is NP-complete is by a reduction from the problem of Exact Cover by 3-Sets.

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

$$\Sigma_{j \in S_i}(3n+1)^{3n-j}$$

and set the target weight and value both to

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

Scheduling

Some examples of the kinds of scheduling tasks that have been proved NP-complete include:

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?

Scheduling

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?

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?

Succinct Certificates

The complexity class NP can be characterised as the collection of languages of the form:

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

Where R is a relation on strings satisfying two key conditions

1. R is decidable in polynomial time.
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 $p(n)$.

Succinct Certificates

y is a *certificate* for the membership of x in L.

Example: If L is SAT, then for a satisfiable expression x, a certificate would be a satisfying truth assignment.

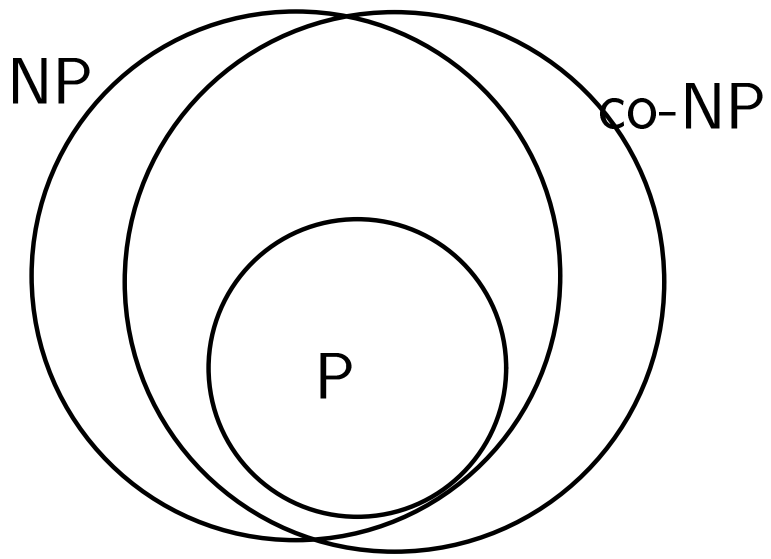
co-NP

As co-NP is the collection of complements of languages in NP, and P is closed under complementation, co-NP can also be characterised as the collection of languages of the form:

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

NP – the collection of languages with succinct certificates of membership.

co-NP – the collection of languages with succinct certificates of disqualification.



Any of the situations is consistent with our present state of knowledge:

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

co-NP-complete

VAL – the collection of Boolean expressions that are *valid* is a *co-NP-complete*.

Any language L that is the complement of an NP-complete language is *co-NP-complete*.

Any reduction of a language L_1 to L_2 is also a reduction of $\bar{L}_1$–the complement of L_1–to $\bar{L}_2$–the complement of L_2.

There is an easy reduction from the complement of SAT to VAL, namely the map that takes an expression to its negation.

$$\mathsf{VAL} \in \mathsf{P} \Rightarrow \mathsf{P} = \mathsf{NP} = \text{co-}\mathsf{NP}$$

$$\mathsf{VAL} \in \mathsf{NP} \Rightarrow \mathsf{NP} = \text{co-}\mathsf{NP}$$

Prime Numbers

Consider the decision problem PRIME:

> Given a number n, is it prime?

This problem is in co-NP.

$$\forall y(y < x \rightarrow (y = 1 \vee \neg(\text{div}(y, x))))$$

> Note, the algorithm that checks for all numbers up to $\sqrt{n}$ whether any of them divides n, is not polynomial, as $\sqrt{n}$ is not polynomial in the size of the input string, which is $\log n$.

Primality

Another way of putting this is that the problem of checking whether a number is composite is in NP.

As it happens, PRIME is also in NP.

Pratt (1976) showed how to construct succinct certificates of primality. They are based on the following number theoretic fact:

> A number $p > 2$ is *prime* if, and only if, there is a number r, $1 < r < p$, such that $r^{p-1} = 1 \bmod p$ and $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for all *prime divisors* q of $p - 1$.

Certificates

A certificate $C(p)$ that p is prime consists of the following:

$$(r, q_1, C(q_1), \ldots, q_k, C(q_k))$$

To complete the proof that PRIME is in NP, we need to prove two things.

1. $C(p)$ is succinct – that is there is a polynomial bound on the length of $C(p)$, in terms of the length of p.
2. There is a polynomial time algorithm that will check that $C(p)$ is a valid certificate of the primality of p.

For the first part, we can show by induction on p that:

$$|C(p)| \leq 4(\log p)^2$$

For the second part, note that $r^k \bmod p$ can be computed with $O(\log k)$ multiplications.

Since each of the multiplications is done $\bmod p$, we know that it involves only $O(\log p)$-bit numbers, and can be done in $O((\log p)^2)$ time.

Finally, $r^k \bmod p$ has to be computed for fewer than $O(\log p)$ distinct numbers k.

Overall, our verification algorithm is $O((\log p)^4)$ – polynomial in the length of p.

Optimisation

The Travelling Salesman Problem was originally conceived of as an optimisation problem

to find a minimum cost tour.

We forced it into the mould of a decision problem – TSP – in order to fit it into our theory of NP-completeness.

Similar arguments can be made about the problems CLIQUE and IND.

This is still reasonable, as we are establishing the *difficulty* of the problems.

A polynomial time solution to the optimisation version would give a polynomial time solution to the decision problem.

Also, a polynomial time solution to the decision problem would allow a polynomial time algorithm for *finding the optimal value*, using binary search, if necessary.

Function Problems

Still, there is something interesting to be said for *function problems* arising from NP problems.

Suppose
$$L = \{x \mid \exists y R(x, y)\}$$
where R is a polynomially-balanced, polynomial time decidable relation.

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.

FNP and FP

A function which, for any given Boolean expression ϕ, gives a satisfying truth assignment if ϕ is satisfiable, and returns "no" otherwise, is a witness function for SAT.

If any witness function for SAT is computable in polynomial time, then $\mathsf{P} = \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then every function in FNP is computable in polynomial time, by a binary search algorithm.

$\mathsf{P} = \mathsf{NP}$ if, and only if $\mathsf{FNP} = \mathsf{FP}$

Under a suitable definition of reduction, the witness functions for SAT are FNP-complete.

Factorisation

The *factorisation* function maps a number n to its prime factorisation:

$$2^{k_1} 3^{k_2} \cdots p_m^{k_m}$$

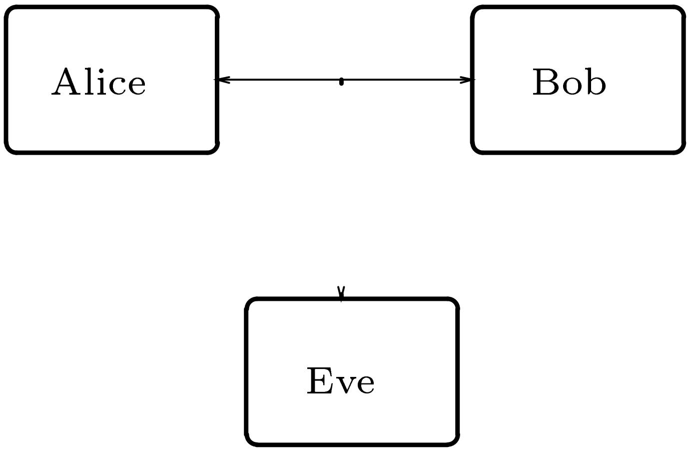
along with certificates of primality for all the primes involved.

This function is in FNP.

The corresponding decision problem (for which it is a witness function) is trivial - it is the set of all numbers.

Still, it is not known whether this function can be computed in polynomial time.

Cryptography



Alice wishes to communicate with Bob without Eve eavesdropping.

Private Key

In a private key system, there are two secret keys

e – the encryption key
d – the decryption key

and two functions D and E such that:

for any x,

$$D(E(x, e), d) = x$$

For instance, taking $d = e$ and both D and E as *exclusive or*, we have the *one time pad*:

$$(x \oplus e) \oplus e = x$$

One Time Pad

The one time pad is provably secure, in that the only way Eve can decode a message is by knowing the key.

If the original message x and the encrypted message y are known, then so is the key:

$$e = x \oplus y$$

Public Key

In public key cryptography, the encryption key e is public, and the decryption key d is private.

We still have,

for any x,

$$D(E(x, e), d) = x$$

If E is polynomial time computable (and it must be if communication is not to be painfully slow), then the function that takes $y = E(x, e)$ to x (without knowing d), must be in FNP.

Thus, public key cryptography is not *provably secure* in the way that the one time pad is. It relies on the existence of functions in FNP – FP.

One Way Functions

A function f is called a *one way function* if it satisfies 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. $f^{-1} \notin \mathsf{FP}$.

We cannot hope to prove the existence of one-way functions without at the same time proving $\mathsf{P} \neq \mathsf{NP}$.

It is strongly believed that the RSA function:

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

is a one-way function.

UP

Though one cannot hope to prove that the RSA function is one-way without separating P and NP, we might hope to make it as secure as a proof of NP-completeness.

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.

UP

Equivalently, UP is the class of languages of the form

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

Where R is polynomial time computable, polynomially balanced, *and* for each x, there is *at most one* y such that $R(x,y)$.

UP One-way Functions

We have

$$\mathsf{P} \subseteq \mathsf{UP} \subseteq \mathsf{NP}$$

It seems unlikely that there are any NP-complete problems in UP.

One-way functions exist *if, and only if,* $\mathsf{P} \neq \mathsf{UP}$.