

Complexity Theory

Lecture 4

Anuj Dawar

University of Cambridge Computer Laboratory
Easter Term 2013

<http://www.cl.cam.ac.uk/teaching/1213/Complexity/>

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 **SAT** \in P?

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

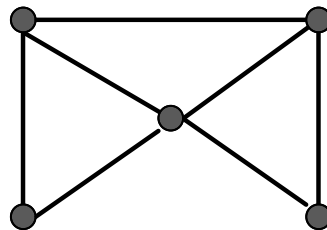
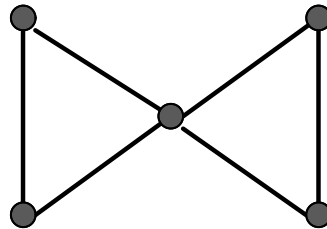
Given a graph $G = (V, E)$, a *Hamiltonian cycle* in G is a path in the graph, starting and ending at the same node, such that every node in V appears on the cycle *exactly once*.

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 numbers less than x ; a truth assignment to the variables of ϕ ; a list of the vertices of G .

The size of the search is *exponential* in the length of the input.

Given a potential solution in the search space, 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.

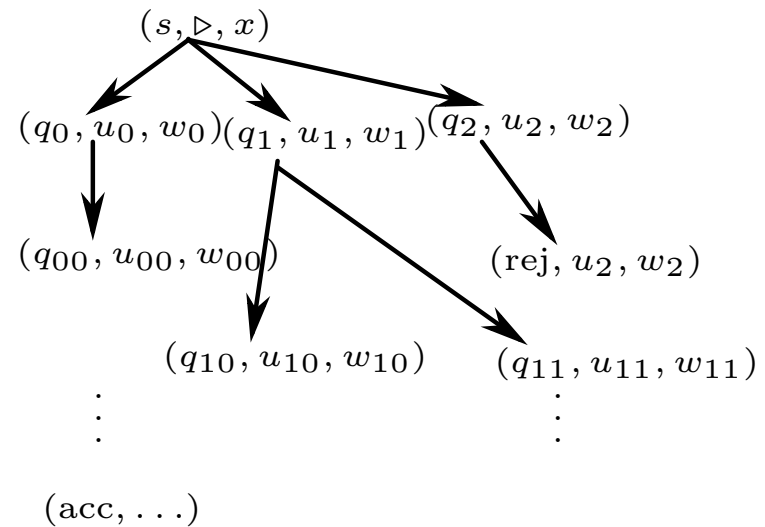
Nondeterministic Complexity Classes

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

$\text{NTIME}(f)$ 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 $f(n)$, where n is the length of x .

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

Nondeterminism



For a language in $\text{NTIME}(f)$, the height of the tree can be 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 .

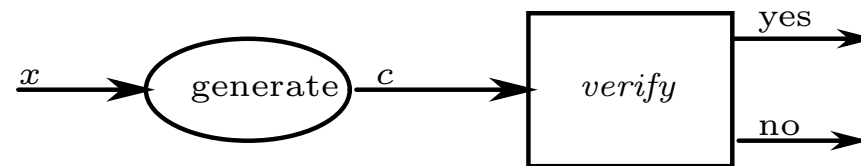
At the i^{th} nondeterministic choice point, V looks at the i^{th} character in c to decide which branch to follow.

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 $L_1 \subseteq \Sigma_1^*$, and $L_2 \subseteq \Sigma_2^*$,

A *reduction* of L_1 to L_2 is a *computable* function

$$f : \Sigma_1^* \rightarrow \Sigma_2^*$$

such that for every string $x \in \Sigma_1^*$,

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

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 $\text{SPACE}(\log n)$, 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 $L_2 \in P$, then $L_1 \in P$

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 $A \in \text{NP}$, $A \leq_P L$.

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