Complexity Theory

Lecture 3: The class NP

Tom Gur

The four stages of learning complexity theory

1) Effortless ignorance
2) Effortful ignorance
3) Effortful knowledge
4) Effortless knowledge

Recap

- Goal: understand the complexity of computational problems
- Strategy: Divide problems into complexity classes
- Post-Turing: Focus on decidable languages.
- Resolution: Polynomial
- Most important class: $\mathcal{P}$ – tractable computation

Today we will go beyond tractable computation!

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.

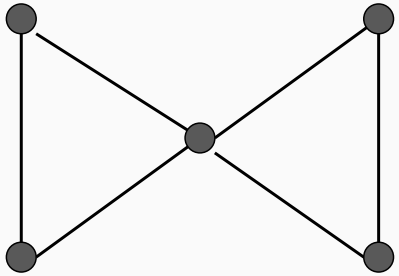
Is Composite $\in$ P?

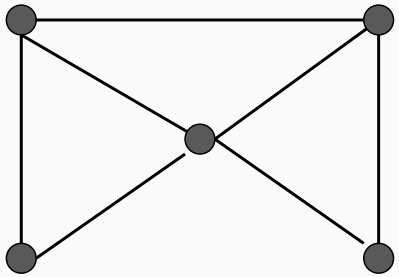
Clearly, the answer is yes if, and only if, Prime $\in$ P.

Is there a conceptual difference between the two?

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*.





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

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 $\mathrm{HAM} \in \mathrm{P}$?

Graph Isomorphism

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is there a *bijection*

$$\pi : V_1 \to V_2$$

such that for every $u, v \in V_1$,

$$(u, v) \in E_1 \quad \text{if, and only if,} \quad (\pi(u), \pi(v)) \in E_2.$$

Is Graph Isomorphism $\in$ P?

Polynomial Verification

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

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

the numbers less than x; truth assignments to the variables of ϕ; lists of the vertices of G; a bijection between V_1 and V_2.

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.

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*.

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

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

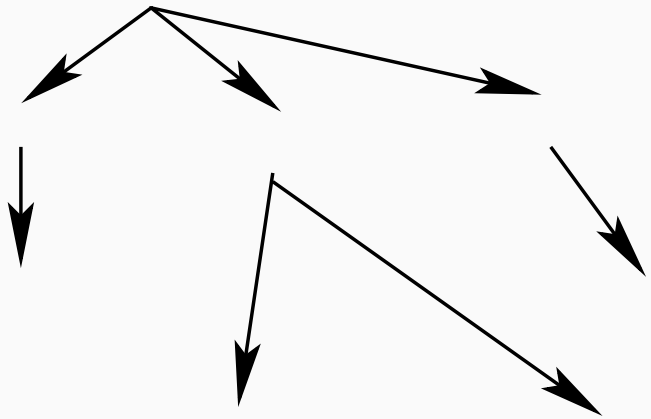
We still define the language accepted by M by:

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

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.



Nondeterministic Complexity Classes

Recall that for any function $f : \mathbb{N} \to \mathbb{N}$, we say that a language L is in $\mathrm{TIME}(f)$ if there is a machine M, such that:

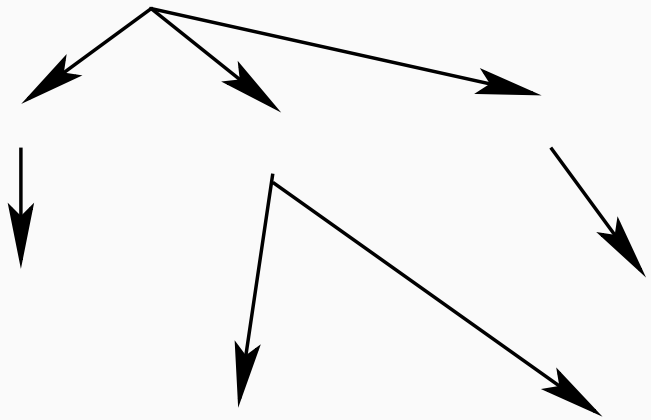
- $L = L(M)$; and
- The running time of M is $O(f)$.

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

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

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

Nondeterminism



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

Turing fact of the day

The problem of *P vs NP* can arguably be traced back to Turing!

2. *Definitions.*

Automatic machines.

If at each stage the motion of a machine (in the sense of § 1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*-machine).

For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in § 1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems. In this

Nondeterminism vs Verification

Theorem
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)

Nondeterminism vs Verification

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.

Why NP and not EXP?