CS:4330 Theory of Computation

Spring 2018

Computability Theory

The class NP

Haniel Barbosa

THE UNIVERSITY OF IOWA

Readings for this lecture

Chapter 7 of [Sipser 1996], 3rd edition. Section 7.3.

Question

Why are we unsuccessful in finding polynomial time algorithms for some problems?

- ▷ Did not try hard enough?
- ▷ Perhaps such problems have, as yet undiscovered, polynomial time algorithms that rely on unknown principles
- ▷ Maybe some of such problems simply cannot be solved in polynomial time. They may be intrinsically difficult.

Trying to answer this central question we start to realize how the complexities of many problems are linked. We further investigate these connections now.

Example

Hamiltonian path problem:

- ▷ A *Hamiltonian path* in a directed graph G is a path that goes through each node of G exactly once
- ▷ Hamiltonian path problem consists of testing whether a directed graph G contains a Hamiltonian path connecting two specified nodes

$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$

It is easy to obtain an exponential algorithm to solve $HAMPATH$, however nobody knows whether a polynomial one exists.

Polynomial Verifiability

The $HAMPATH$ problem has a feature called *polynomial verifiability* which is important for understanding its complexity:

> *If a Hamiltonian path in a graph G is discovered (no matter how) we could easily convince someone else of its existence, simply by presenting it!*

This is, *verifying* the existence of a Hamiltonian graph may be much easier than *determining* its existence.

Another example

- ▷ Another example of polynomially verifiable problem is number compositeness.

- ▷ A natural number is *composite* if it is not prime. Let

$$COMPOSITES = \{x \mid x \in \mathbb{N} \text{ and } x = pq \text{ s.t. } p, q \in \mathbb{N},\ p > 1 \text{ and } q > 1\}$$

It can easily be verified that a number x is composite, all that is needed is a divisor of x

Note

- ▷ There are problems that may not be polynomially verifiable
- ▷ For example, $\overline{HAMPATH}$, the complement of $HAMPATH$, is not polynomial time verifiable
- ▷ Even if we could determine that a graph did not have a Hamiltonian path we don't know a way for verifying its nonexistence without using the same exponential time algorithm that determined its nonexistence!

Verifier

Definition

A *verifier* for a language A is a an algorithm V such that

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w, so a *polymial time verifier* runs in polynomial time in the length of w. A language A is *polynomially verifiable* if it has a polynomial time verifier.

- ▷ A verifier uses additional information, represented by c in the definition
- ▷ This information is called a *certificate*, or *proof*, of membership
- ▷ The certificate must have polynomial length w.r.t. the length of w
- ▷ For example, the certificate for $\langle G, s, t \rangle \in HAMPATH$ is a Hamiltonian path between s and t; the certificate for $x \in COMPOSITES$ is a divisor p of x

The class NP

Definition

NP is the class of languages that have polynomial time verifiers.

- ▷ The term **NP** comes from *nondeterministic polynomial time* and is derived from an alternative characterization using nondeterministic polynomial time Turing machines
- ▷ **NP** class is important because it contains many problems of practical interest. For example $HAMPATH, COMPOSITES \in \mathbf{NP}$
- ▷ Moreover, $COMPOSITES \in \mathbf{P}$, but proving it is more difficult

An NTM deciding $HAMPATH$

$N_1 =$"On input string $\langle G, s, t \rangle$ where G is a directed graph containing nodes s, t:

1. Write a list of m numbers $p_1, \ldots, p_m$ where m is the number of nodes in G. Each p_i is nondeterministically selected between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*
3. Check whether $s = p_1$ and $t = p_m$. If either fails, *reject*
4. For each i, $1 \leq i \leq m$, check whether p_1, p_{i+1} is an edge of G. If any are not, *reject*. Otherwise, *accept*."

Each stage clearly runs ins nondeterministic polynomial time, thus so does N_1.

Redefining NP in terms of NTMs

Theorem

*A language is in **NP** iff it is decided by some nondeterministic polynomial time Turing machine.*

Proof idea

- ▷ We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice-versa
- ▷ The verifier simulates the NTM by using the accepting branches as certificates

Proof

- ▷ For the forward direction we consider a language $A \in \mathbf{NP}$ and show that A is decidable by a polynomial time NTM
- ▷ Assume V is a polynomial time verifier deciding A in time n^k. The NTM N_V equivalent to V works as follows:
  $N_V =$"On input string w of length n:
  1. Nondeterministically select a string c of length at most n^k
  2. Run V on input $\langle w, c \rangle$
  3. If V accepts, *accept*. Otherwise, *reject*."
- ▷ For the other direction, assume that A is decided by a polynomial time NTM and construct a polynomial verifier V_N that decides A:
  $V_N =$"On input $\langle w, c \rangle$ where w and c are strings:
  1. Simulate N on input w using each symbol of c as a description of the nondeterministic choice to make at each step (see NTM computation simulation)
  2. If this branch of N's computation accepts, *accept*. Otherwise, *reject*."

Class $\mathrm{NTIME}(t(n))$

Definition

The nondeterministic time complexity class $\mathrm{NTIME}(t(n))$ is defined by

$$\mathrm{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } \mathcal{O}(t(n)) \text{ NTM}\}$$

With this definition a corollary of the previous theorem is that

Corollary

$$\textbf{\textit{NP}} = \bigcup_k \mathrm{NTIME}(n^k)$$

Observations

- ▷ The class **NP** is insensitive to the choice of reasonable nondeterministic computation model because all such models are polynomially equivalent
- ▷ When describing and analyzing nondeterministic polynomial time algorithms we follow the notation conventions set up for deterministic polynomial time algorithms
- ▷ Each must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model
- ▷ Algorithm analysis must show that every branch uses at most polynomially many stages

Example: Undirected graphs and cliques

- ▷ A *clique* in an undirected graph is a subgraph wherein every two nodes are connected by an edge
- ▷ A k-clique is a clique containing k nodes

The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$$

Theorem

$CLIQUE \in \mathbf{NP}$

Example: Undirected graphs and cliques

▷ Proof idea:
The clique is the certificate. The following is a verifier V for $CLIQUE$:

$V =$"On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, *accept*. Otherwise, *reject*."

▷ Alternative proof idea:
If one prefers to think in terms of polynomial time NTM one can construct the following NTM N that decides $CLIQUE$:

$N =$"On input $\langle G, k\rangle$ where G is an undirected graph:

1. Nondeterministically select a subset C of k nodes of G
2. Test whether all nodes in c are connected and whether G contains all edges connecting nodes in C
3. If yes, *accept*. Otherwise, *reject*."

Example: *SUBSET-SUM* problem

A collection of k integers $x_1, \ldots, x_k$ and a target number t are given. We want to determine whether this collection contains a subcollection that adds up to t

$$SUBSET\text{-}SUM = \left\{ \langle S, t \rangle \;\middle|\; \begin{array}{l} S = \{x_1, \ldots, x_k\} \text{ and for some} \\ C = \{y_1, \ldots, y_l\} \subseteq S \text{ we have} \\ \sum_{i=1}^{i=l} y_i = t \end{array} \right\}$$

Example: $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in SUBSET\text{-}SUM$ since $4 + 21 = 25$

$\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ may be multisets (i.e. elements may be repeated)

Theorem

$SUBSET\text{-}SUM \in \mathbf{NP}$

Example: *SUBSET-SUM* problem

▷ Proof idea:
the subset is the certificate. The following is a verifier V for *SUBSET-SUM*:

$V =$"On input $\langle \langle S, t \rangle, c \rangle$:

1. Test whether c is a collection of numbers that sum to t
2. Test whether S contains all the numbers in C
3. If both pass, *accept*. Otherwise, *reject*."

▷ Alternative proof idea:
We can also prove this theorem given the NTM N that decides *SUBSET-SUM*:

$N =$"On input $\langle S, t \rangle$:

1. Nondeterministically select a subset C of numbers in S
2. Test whether C is a collection of numbers that sum to t
3. If the test passes, *accept*. Otherwise, *reject*."

Observations

- ▷ Generally, verifying that something is *not* present is more difficult than that it is present
- ▷ Note that $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$ are not obviously members of **NP**
- ▷ A separate complexity class denoted **coNP** contains all languages that are complements of languages in **NP** class
- ▷ It is an open question whether **NP** is different from **coNP**