CS301: Computability and Complexity Theory (CC)

Lecture 10: Class NP

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

December 15, 2023

Table of contents

Section 1

Previously on CS301

Class P

The most important definition in complexity theory

Definition

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. In other words

$$P = \bigcup_k TIME(n^k)$$

The class P plays a central role in our theory and is important because

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM (robustnes)
- P roughly corresponds to the class of problems that are realistically solvable on a computer (relevance)

Examples

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things:

1. First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n
2. Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. We choose the stages when we describe the algorithm to make this second part of the analysis easy to do

When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial

Examples

- One point that requires attention is the encoding method used for problems
- We continue to use the angle-bracket notation $< . >$ to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method
- A reasonable method is one that allows for polynomial time encoding and decoding of objects into natural internal representations or into other reasonable encodings
- For example, many computational problems we study next contain encodings of graphs
- One reasonable encoding of a graph is a list of its nodes and edges. Another is the adjacency matrix, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not
- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation
- In reasonable graph representations, the size of the representation is a polynomial in the number of nodes
- Thus, if we analyze an algorithm and show that its running time is polynomial (or exponential) in the number of nodes, we know that it is polynomial (or exponential) in the size of the input

Section 2

Context setup

Context setup

Corresponding to 7.3

Context setup

- We continue the study of **P**
- Next we move to **NP**

Section 3

Class P

Example II

Let's study another example of a polynomial time algorithm. Say that two numbers are relatively prime if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself, whereas 10 and 22 are not relatively prime.

Let *RELPRIME* be the problem of testing whether two numbers are relatively prime. Let

$$RELPRIME = \{< x, y > \mid x \text{ and } y \text{ are relatively prime}\}$$

Theorem

$RELPRIME \in P$

Example II

Proof idea: One algorithm that solves this problem searches through all possible divisors of both numbers and accepts if none are greater than 1. However, the magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation.
Instead, we solve this problem with an ancient numerical procedure, called the *Euclidean algorithm*, for computing the greatest common divisor. The *greatest common divisor* of natural numbers x and y, written $gcd(x, y)$, is the largest integer that evenly divides both x and y. Obviously, x and y are relatively prime iff $gcd(x, y) = 1$.

Example II

Proof.

The Euclidean algorithm E is as follows

E = On input $< x, y >$, where x and y are natural numbers in binary:

```
1. Repeat until $y = 0$:
2.     Assign $x \leftarrow x \, mod \, y$
3.     Exchange $x$ and $y$
4. Output $x$
```

Algorithm R solves $RELPRIME$, using E as a subroutine

R = On input $< x, y >$ where x and y are natural numbers in binary:

```
1. Run $E$ in $< x, y >$
2. If the result is 1, accept. Otherwise, reject.
```

Example II

Proof.

Clearly, if E runs correctly in polynomial time, so does R and hence we only need to analyze E for time and correctness. The correctness of this algorithm is well known so we won't discuss it further.

To analyze the time complexity of E, we first show that every execution of stage 2 (except possibly the first) cuts the value of x by at least half. After stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently executed, $x > y$. If $x/2 \geq y$, then $x \; mod \; y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $x \; mod \; y = x - y < x/2$ and x drops by at least half.

Example II

Proof.

The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop. Thus, the maximum number of times that stages 2 and 3 are executed is the lesser of $2log_2x$ and $2log_2y$. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as $O(n)$. Each stage of E uses only polynomial time, so the total running time is polynomial. $\square$

Example III

Another example of a polynomial time algorithm shows that every context-free language is decidable in polynomial time

Theorem

Every context-free language is a member of P

Proof idea: We proved that every CFL is decidable. To do so, we gave an algorithm for each *CFL* that decides it. If that algorithm runs in polynomial time, the current theorem follows as a corollary. Let's recall that algorithm and find out whether it runs quickly enough.

Let L be a *CFL* generated by *CFG* G that is in Chomsky normal form. A derivation of a string w has $2n - 1$ steps, where n is the length of w because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n - 1$ steps when its input is a string of length n. If any of these is a derivation of w, the decider accepts; if not, it rejects.

Example III

A quick analysis of this algorithm shows that it doesn't run in polynomial time. The number of derivations with k steps may be exponential in k, so this algorithm may require exponential time.

To get a polynomial time algorithm, we introduce a powerful technique called **dynamic programming**. This technique uses the accumulation of information about smaller subproblems to solve larger problems. We record the solution to any subproblem so that we need to solve it only once. We do so by making a table of all subproblems and entering their solutions systematically as we find them.

In this case, we consider the subproblems of determining whether each variable in G generates each substring of w. The algorithm enters the solution to this subproblem in an $n \times n$ table. For $i \leq j$, the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \dots w_j$. For $i > j$, the table entries are unused.

Example III

The algorithm fills in the table entries for each substring of w. First it fills in the entries for the substrings of length 1, then those of length 2, and so on. It uses the entries for the shorter lengths to assist in determining the entries for the longer lengths.

For example, suppose that the algorithm has already determined which variables generate all substrings up to length k. To determine whether a variable A generates a particular substring of length $k + 1$, the algorithm splits that substring into two nonempty pieces in the k possible ways. For each split, the algorithm examines each rule $A \to BC$ to determine whether B generates the first piece and C generates the second piece, using table entries previously computed. If both B and C generate the respective pieces, A generates the substring and so is added to the associated table entry. The algorithm starts the process with the strings of length 1 by examining the table for the rules $A \to b$.

Example III

Proof.

The following algorithm D implements the proof idea. Let G be a CFG in Chomsky normal form generating the CFL L. Assume that S is the start variable. (Recall that the empty string is handled specially in a Chomsky normal form grammar. The algorithm handles the special case in which $w = \epsilon$ in stage 1)

```
D = On input w = w_1 w_2 ... w_n
1. For $w = \epsilon$, if $S \to \epsilon$ is a rule, accept; else, reject [$w = \epsilon$ case]
2. For $i = 1 \to n$: [examine each substring of length 1]
3.     For each variable $A$:
4.         Test whether $A \to b$ is a rule, where $b = w_i$
5.         If so, place $A$ in $table(i, i)$
```

Example III

Proof.

```
6.  For $l = 2 \to n$: [$l$ is the length of the substring]
7.      For $i = 1 \to n - l + 1$ [$i$ is the start position of the substring]
8.          Let $j = i + l - 1$ [$j$ is the end position of the substring]
9.          For $k = i \to j - 1$: [$k$ is the split position]
10.             For each rule $A \to BC$:
11.                 If $\textit{table}(i, k)$ contains $B$ and $\textit{table}(k + 1, j)$ contains $C$, put $A$ in $\textit{table}(i, j)$
12. If $S$ is in $\textit{table}(1, n)$, *accept* ; else, *reject*
```

Example III

Proof.

Now we analyze D. Each stage is easily implemented to run in polynomial time. Stages 4 and 5 run at most nv times, where v is the number of variables in G and is a fixed constant independent of n; hence these stages run $O(n)$ times.

Stage 6 runs at most n times. Each time stage 6 runs, stage 7 runs at most n times. Each time stage 7 runs, stages 8 and 9 run at most n times.

Each time stage 9 runs, stage 10 runs r times, where r is the number of rules of G and is another fixed constant. Thus stage 11, the inner loop of the algorithm, runs $O(n^3)$ times. Summing the total shows that D executes $O(n^3)$ stages $\square$

Section 4

Class NP

Class NP

As we previously observed, we can avoid brute-force search in many problems and obtain polynomial time solutions. However, attempts to avoid brute force in certain other problems, including many interesting and useful ones, haven't been successful, and polynomial time algorithms that solve them aren't known to exist

Why have we been unsuccessful in finding polynomial time algorithms for these problems? We don't know the answer to this important question. Perhaps these problems have as yet undiscovered polynomial time algorithms that rest on unknown principles. Or possibly some of these problems simply cannot be solved in polynomial time. They may be intrinsically difficult

One remarkable discovery concerning this question shows that the complexities of many problems are linked. A polynomial time algorithm for one such problem can be used to solve an entire class of problems. To understand this phenomenon, let's begin with an example

Example I

A *Hamiltonian path* in a directed graph G is a directed path that goes through each node exactly once. We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes, as shown in the following figure. Let

$$HAMPATH = \{< G, s, t > \mid G \textit{ is a directed graph with a Hamiltonian path from s to t}\} \quad (1)$$

We can easily obtain an exponential time algorithm for the *HAMPATH* problem by modifying the brute-force algorithm for *PATH*. We need only add a check to verify that the potential path is Hamiltonian. No one knows whether *HAMPATH* is solvable in polynomial time.

Example I

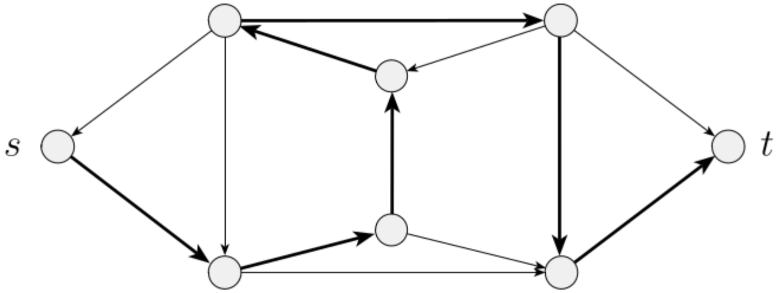

Figure: A Hamiltonian path goes through every node exactly once

Example I

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

Even though we don't know of a fast (i.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence simply by presenting it

In other words, verifying the existence of a Hamiltonian path may be much easier than determining its existence

Example II

Another polynomially verifiable problem is *compositeness*. Recall that a natural number is composite if it is the product of two integers greater than 1. Let

$$COMPOSITES = \{x \mid x = pq \text{ for integers } p, q > 1\}$$

We can easily verify that a number is composite—all that is needed is a divisor of that number. Recently (2002 :)), a polynomial time algorithm for testing whether a number is prime or composite was discovered, Called *AKS primality test* it is considerably more complicated than the preceding method for verifying compositeness and runs in $O(n^{10})$

Verifiers

Some problems may not be polynomially verifiable. For example, take $\overline{HAMPATH}$, the complement of the $HAMPATH$ problem

Even if we could determine (somehow) that a graph did not have a Hamiltonian path, we don't know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place

Verifiers

Definition

A **verifier** for a language A is an algorithm V , where

$$A = \{w \mid V \textit{ accepts } < w, c > \textit{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of w, so a *polynomial 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 the symbol c, to verify that a string w is a member of A. This information is called a *certificate*, or *proof*, of membership in A
Observe that for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound

Verifiers

- For the *HAMPATH* problem, a certificate for a string $< G, s, t > \in HAMPATH$ simply is a Hamiltonian path from s to t
- For the COMPOSITES problem, a certificate for the composite number x simply is one of its divisors
- In both cases, the verifier can check in polynomial time that the input is in the language when it is given the certificate

Class NP

Definition

NP is the class of languages that have polynomial time verifiers

The class NP is important because it contains many problems of practical interest. From the preceding discussion, both *HAMPATH* and *COMPOSITES* are members of NP

The term NP comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. Problems in NP are sometimes called NP-problems

Example I

The following is a nondeterministic TM (*NTM*) that decides the *HAMPATH* problem in nondeterministic polynomial time (we defined the time of a nondeterministic machine to be the time used by the longest computation branch)

$N_1 =$ On input $< G, s, t >$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be 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 fail, *reject*
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*

Example I

To analyze this algorithm and verify that it runs in nondeterministic polynomial time, we examine each of its stages:

- In stage 1, the nondeterministic selection clearly runs in polynomial time
- In stages 2 and 3, each part is a simple check, so together they run in polynomial time
- In stage 4 also clearly runs in polynomial time
- Thus, this algorithm runs in nondeterministic polynomial time

Class NP

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time TM

Proof idea: We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate

Class NP

Proof.

For the forward direction of this theorem, let $A \in NP$ and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows

$N =$ On input w of length n

1. Nondeterministically select string c of length at most n^k
2. Run V on input $< w, c >$
3. If V accepts, *accept* ; otherwise, *reject*

Class NP

Proof.

To prove the other direction of the theorem, assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows

$V =$ On input $< w, c >$, where w and c are strings

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*

$\square$

NTIME

We define the nondeterministic time complexity class $NTIME(t(n))$ as analogous to the deterministic time complexity class $TIME(t(n))$

Definition

$NTIME(t(n)) = \{L \mid L \textit{ is a language decided by an } O(t(n)) \textit{ time nondeterministic } TM\}$

Corollary

$NP = \bigcup_k NTIME(n^k)$

Class NP

- The class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent
- When describing and analyzing nondeterministic polynomial time algorithms, we follow the preceding conventions for deterministic polynomial time algorithms
- Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model
- We analyze the algorithm to show that every branch uses at most polynomially many stages

Example II

A clique in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A *k-clique* is a clique that contains k nodes

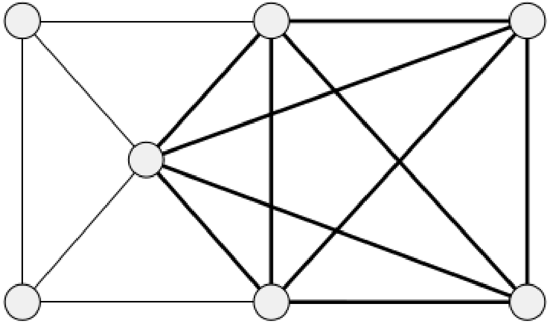

Figure: A graph with a 5-clique

Example II

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

$$CLIQUE = \{< G, k > \mid G \textit{ is an undirected graph with a } k - \textit{clique}\}$$

Theorem

$CLIQUE \in NP$

Proof idea: The clique is the certificate

Example II

Proof.

The following is a verifier V for $CLIQUE$

$V =$ On input $<< G, k >, c >$:

1. Test whether c is a subgraph with k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, *accept* ; otherwise, *reject*

$\square$

Example II

Proof.

If you prefer to think of NP in terms of nondeterministic polynomial time TM, you may prove this theorem by giving one that decides *CLIQUE*

$N =$ On input $< G, k >$, where G is a graph

1. Nondeterministically select a subset c of k nodes of G
2. Test whether G contains all edges connecting nodes in c
3. If yes, *accept* ; otherwise, *reject*

$\square$

Example III

Next, we consider the *SUBSETSUM* problem concerning integer arithmetic. We are given a collection of numbers $x_1, \ldots, x_k$ and a target number t. We want to determine whether the collection contains a subcollection that adds up to t

$$\begin{aligned} SUBSETSUM = \{< S, t > \mid S = \{x_1, \ldots, x_k\} & \textit{ and} \\ \textit{for some } \{y_1, \ldots, y_l\} \subset \{x_1, \ldots, x_k\} & \textit{ we have } \sum y_i = t\} \end{aligned}$$

For example, $< \{4, 11, 16, 21, 27\}, 25 > \in SUBSETSUM$ because $4 + 21 = 25$. Note that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are considered to be multisets and so allow repetition of elements

Example III

Theorem

SUBSETSUM $\in$ *NP*

Proof.

The following is a verifier V for SUBSETSUM

$V =$ On input $<< S, t >, c >$:

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*

$\square$

Example III

Proof.

We can also prove this theorem by giving a nondeterministic polynomial time TM for SUBSETSUM as follows:

$N =$ On input $< S, t >$

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

$\square$

coNP

Observe that the complements of these sets, $\overline{CLIQUE}$ and $\overline{SUBSETSUM}$, are not obviously members of NP. Verifying that something is not present seems to be more difficult than verifying that it is present. We make a separate complexity class, called **coNP**, which contains the languages that are complements of languages in NP

We don't know whether $coNP \neq NP$

Section 5

P vs NP

P vs NP

As we saw, NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine; or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time

P is the class of languages where membership can be tested in polynomial time. We summarize this information as follows, where we loosely refer to polynomial time solvable as solvable "quickly"

P = the class of languages for which membership can be *decided* quickly

NP = the class of languages for which membership can be *verified* quickly

P vs NP

- We have presented examples of languages, such as HAMPATH and CLIQUE, that are members of NP but that are not known to be in P
- The power of polynomial verifiability seems to be much greater than that of polynomial decidability
- But, hard as it may be to imagine, P and NP could be equal
- We are unable to prove the existence of a single language in NP that is not in P
- The question of whether P $=$ NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics

P vs NP

- The question of whether P $=$ NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics
- If these classes were equal, any polynomially verifiable problem would be polynomially decidable
- Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success
- Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search
- Doing so is presently beyond scientific reach

P vs NP

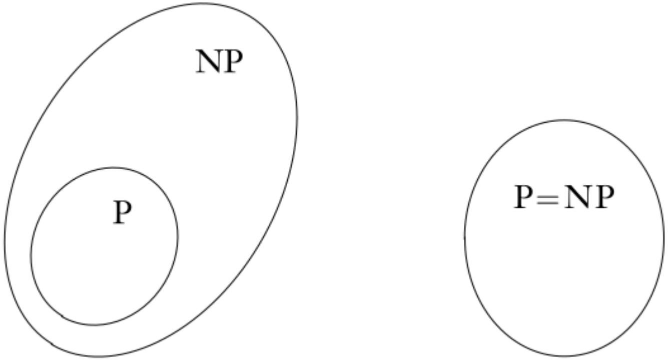

Figure: One of these two possibilities is correct

P vs NP

The best deterministic method currently known for deciding languages in NP uses exponential time. In other words, we can prove that

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$

but we don't know whether NP is contained in a smaller deterministic time complexity class

P vs NP

https://www.win.tue.nl/~wscor/woeginger/P-versus-NP.htm