Lecture 21: P & NP

Complexity Classes

The Theory of Complexity deals with the classification of decision problems (languages) into several classes:

- the class of "easy" problems,
- the class of "hard" problems,
- the class of "harder" problems;
- the class of "harderer" problems;
- . . . ;
- the class of "hardest" problems;
- the class of unsolvable problems.

Question

How to classify the problems?

Answer: polynomial-time (and polynomial-space, which we won't cover).

Polynomial-Time Algorithms

Definition

An algorithm (TM) is polynomial-time if its running time (number of steps) is $O(n^c)$ for any input x, where c is a constant independent of n, and $n = |x|$ is the input size. Otherwise, the algorithm (TM) is non-polynomial time.

Remark:

- We can use any of the deterministic variants of the Turing machine as the computational model.
- When the input consists of integers, we may also use the number of integers as the input size n, provided that it differs only polynomially from the real input size. (Pay attention to the case where there input consists of only a constant number of integers.)

Polynomial-Time Algorithms

Example (Polynomial-time algorithms)

- The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m_1 and m_2 are, respectively, the number of digits in the two integers.
- Given a DFA M and an input string w, determine if M accepts w.
- (Almost) all algorithms you learned in COMP 3711.

Example (Non-polynomial-time algorithms)

- Given an NFA M and a string w, the algorithm learned in class that determines if M accepts w.
- For a fixed nondeterministic TM M that halts on all inputs in polynomial time, the algorithm learned in class that, given an input string w, decide if M accepts w.
- The naive algorithm that checks if a given integer is prime.

Blame yourself.

I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

Show that no-efficient algorithm exists
(like showing a problem is unsolvable)

I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

The "Completeness" Approach

Unfortunately, for many hard but solvable problems that occur in practice, we still cannot formally prove that they are hard (i.e., do not have polynomial-time algorithms).
We will learn about NP-Completeness, which gives us strong evidence that these problems are hard.

- There is a very large class of thousands of practical problems for which it is not known if they have polynomial-time algorithms.
- It is known that if any one of these NP-complete problems has a polynomial-time algorithm then all of them have polynomial-time algorithms.
- Researchers have spent innumerable man-years trying to find polynomial-time algorithms to these problems and failed.
- So, NP-complete problems are very likely to be hard.
- What do you do: prove that your problem is NP-complete.

The "Completeness" Approach

What do you actually do:

I couldn't find a polynomial-time algorithm,
but neither could all these other smart people!

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding simpler decision problem.
- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.
- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.
- Since decision problems correspond to languages, they are easy to relate and compare.
- In fact, if the decision problem can be solved efficiently, the corresponding optimization problem can usually be solved efficiently as well.

Decision Problem: MST

Optimization problem: Minimum Spanning Tree

Given a weighted graph G, find a minimum spanning tree (MST) of G.

Decision problem: Decision Spanning Tree (DST)

Given a weighted graph G and an integer k, does G have a spanning tree of weight at most k?

- The inputs are of the form (G, k).
- Following the notation of languages, we will write $(G, k) \in DST$ or $(G, k) \notin DST$ to denote, respectively, yes and no answers.

Decision Problem: Knapsack

We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, ..., w_n$ and values $v_1, ..., v_n$, where v_i and w_i are positive integers.

Optimization problem: Knapsack

Find the largest value $\sum_{i \in T} v_i$ of any subset T that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.

Decision problem: Decision Knapsack (DKnapsack)

Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Is the Knapsack Algorithm Polynomial?

Recall the algorithm you learned in COMP 3711.
We can either leave the ith item or to take it. If we take it, we gain value v_i, but only $w - w_i$ space remains. Thus

$$V[i, w] = \max(V[i-1, w],\ v_i + V[i-1, w - w_i])$$

Then we evaluate this recurrence using dynamic programming for all $i = 0, \ldots, n$ and $w = 0, \ldots, W$. The running time is $O(nW)$.

Is the Knapsack Algorithm Polynomial?

Answer: No!

- The size of the input is

$$size(I) = \log_2 W + \sum_i \log_2 w_i + \sum_i \log_2 v_i.$$

- nW is not polynomial in $size(I)$. Depending upon the values of the w_i and v_i, nW could even be exponential in $size(I)$.

It is unknown as to whether there exists a polynomial time algorithm for Knapsack.

- In fact, Knapsack is an **NP**-Complete problem.

The Class **P**

Definition

The class **P** consists of all decision problems (languages) that are solvable in polynomial time. That is, there exists an algorithm that decides in polynomial time if any given input is a yes-input or a no-input.

Question

How to prove that a decision problem is in **P**?

Ans: You need to find a polynomial-time algorithm for this problem.

Question

How to prove that a decision problem is not in **P**?

Ans: You need to prove there is no polynomial-time algorithm for this problem (much harder).

Closure Properties of **P**

Theorem

P *is closed under complement, union, intersection, concatenation, and Kleene star.*

Proof.

The proof for the first four operations is essentially the same as that for recursive languages.
For Kleene star, use dynamic programming. □

The Class **NP**

Definition

A nondeterministic TM runs in polynomial time if for any input x, the number of steps of any computation path is $O(n^c)$, where c is a constant and $n = |x|$ is the input size. The class **NP** consists of all decision problems that can be decided by a nondeterministic TM in polynomial time.

Remark: **NP** stands for "nondeterministic polynomial time", not "non-polynomial"!

Certificates and Verifying Certificates

Definition

A certificate is a string corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

- By definition, only yes-input needs a certificate (a no-input does not need to have a 'certificate' to show it is a no-input).

Verifying a certificate: There is an algorithm that, given a presumed yes-input and its corresponding certificate, by making use of the given certificate, verifies that the input is indeed a yes-input.

An Alternative Definition of **NP**

Theorem

A decision problem belongs to **NP** *iff for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.*

Proof.

"only if": Let M be a nondeterministic TM deciding the problem. The certificate simply consists of all the correct choices in all the nondeterministic steps of M that leads to a "yes" final state. Then we can simulate M following the advice of the certificate and verify that the final state is "yes".

"if": Let M be a deterministic TM that does the verification job. Suppose M runs in time $p(n)$ for some polynomial p. Then there must exist a certificate with length at most $p(n)$. We build a nondeterministic TM that first nondeterministically enumerates all strings of length at most $p(n)$, then simulates M with that certificate, and reach a "yes" final state if the verification succeeds. □

COMPOSITE $\in$ **NP**

COMPOSITE

Is a given positive integer n composite?

- For COMPOSITE, an yes-input is just the integer n that is composite.

Question (Certificate)

What is needed to show n (a presumed yes-input) is actually a yes-input? The 'object' needed is the certificate for COMPOSITE.

Ans: The certificate is an integer a $(1 < a < n)$ with the property that it divides n.

Proof (Verifying a certificate).

- Given a certificate a, check whether a divides n.
- This can be done in time $O((\log_2 n)^2)$ (recall that input size is $\log_2 n$ so this is polynomial in input size).
- Hence, COMPOSITE $\in$ **NP**. □

DSubsetSum $\in$ NP

DSubsetSum

Input is a positive integer C and n positive integers $s_1, ..., s_n$. Is there a subset of these integers that add up to exactly C?

Example

$\{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $C = 138457$
Subset: $\{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$

- A DSubsetSum yes-input consists of n numbers, and an integer C, such that there is a subset of those integers that add up to C.

DSubsetSum $\in$ **NP**...

Question (Certificate)

What is needed to show that the given input is actually a yes-input?

Ans: A subset T of subscripts with the corresponding integers add up to C.

Proof (Verifying a certificate).

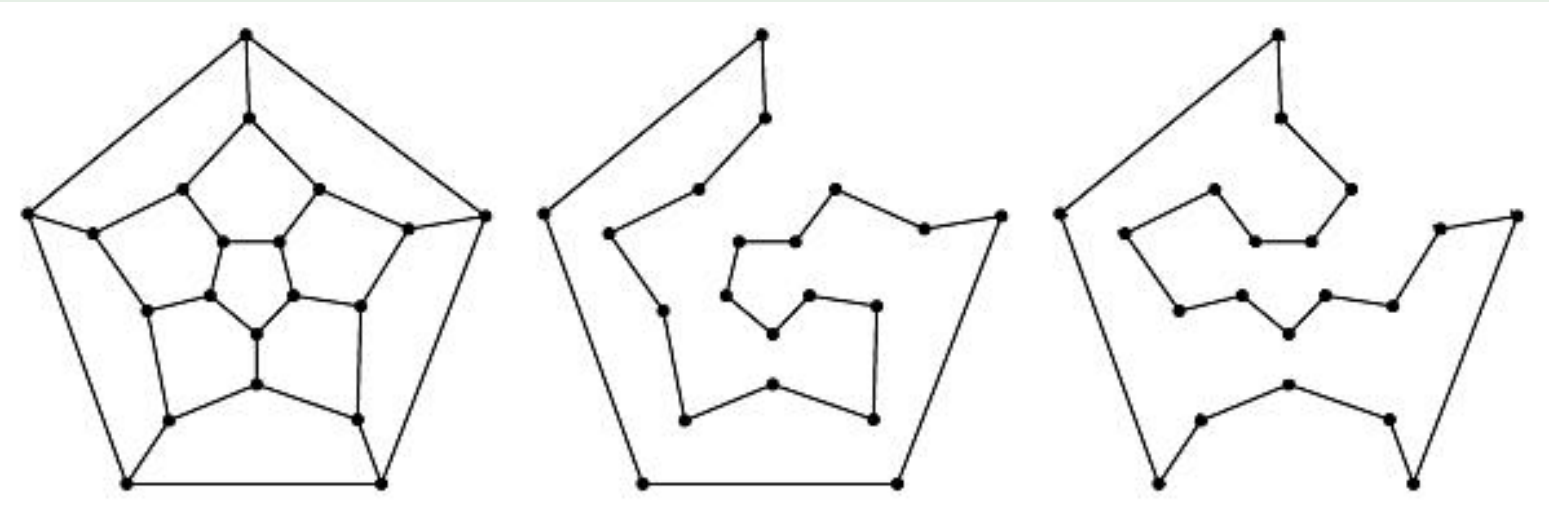
- Given a subset T of subscripts, check whether $\sum_{i \in T} s_i = C$.
- Input-size is $m = (\log_2 C + \sum_{i=1}^{n} \log_2 s_i)$ and verification can be done in time $O(\log_2 C + \sum_{i \in T} \log_2 s_i) = O(m)$, so this is polynomial time.
- Hence we have DSubsetSum $\in$ **NP**. $\square$

DHamCyc $\in$ **NP**

Definition (Hamiltonian Cycle)

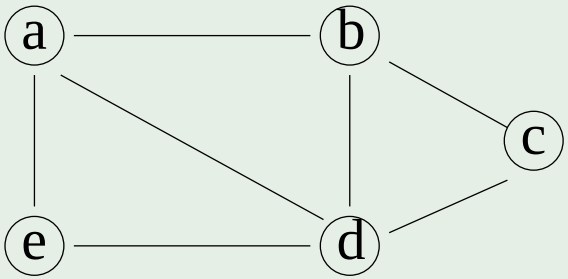
Input is a graph $G = (V, E)$. A cycle of graph G is called Hamiltonian if it contains every vertex exactly once.

Example



DHamCyc $\in$ **NP**...

Example



Find a Hamiltonian cycle for this graph

Optimization problem: HamCyc

Find a Hamiltonian cycle for this graph or say that one doesn't exist.

Decision problem: DHamCyc

Does G have a Hamiltonian cycle?

Certificate: An ordering of the n vertices in G (corresponding to their order along the Hamiltonian cycle), i.e., $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$.

DHamCyc $\in$ **NP**

Proof (Verifying a certificate).

- Given a certificate, the verification algorithm checks whether it is a Hamiltonian cycle of G by simply checking whether all of the edges $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1})$ appear in the graph.
- This can be done in $O(n)$ time so this is polynomial in input size.
- Hence, DHamCyc $\in$ **NP**. □

Important Note:

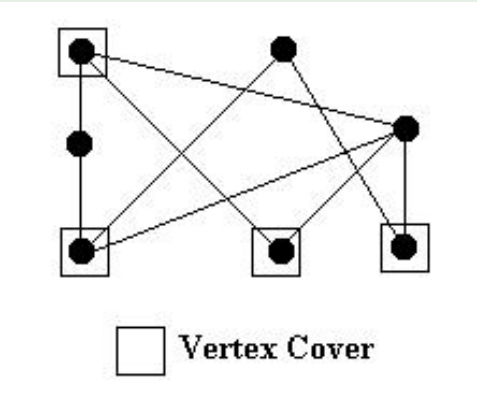
- There are many possible types of certificates for DHamCyc. This is only one such.
- Another type of certificate might be a set of n edges for which it has to be confirmed that they are all in G and that they form a Hamiltonian Cycle.

DVC $\in$ NP

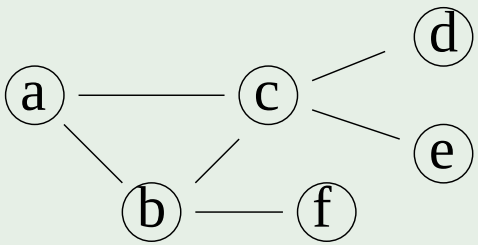
Definition (Vertex Cover)

A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.

Example



Example



Find a vertex cover of G
of size two

DVC $\in$ **NP**...

Decision Vertex Cover (DVC) Problem

Given an undirected graph G and an integer k, does G have a vertex cover with k vertices?

Claim

$DVC \in \mathbf{NP}$.

Proof.

- A certificate will be a set C of k vertices.
- The brute force method to check whether C is a vertex cover takes time $O(ke)$. As $ke < (n+e)^2$, the time to verify is $O((n+e)^2)$. So a certificate can be verified in polynomial time.

Closure Properties of **NP**

Theorem

NP *is closed under union, intersection, concatenation and Kleene star.*

Proof.

Similar to the proof for recursive languages. □

Question

Is **NP** closed under complement?

Ans: Unknown.

Note: The class $\{\overline{L} \mid L \in \mathbf{NP}\}$ is denoted **coNP**. It is still an open problem whether $\mathbf{NP} = \mathbf{coNP}$. The belief is that $\mathbf{NP} \neq \mathbf{coNP}$.

Satisfiability I

We will now introduce Satisfiability (SAT), which, we will see later, is one of the most important **NP** problems.

Definition

A Boolean formula is a logical formula consisting of

1. Boolean variables (0=false, 1=true),
2. logical operations
   - $\bar{x}$: NOT,
   - $x \vee y$: OR,
   - $x \wedge y$: AND.

These are defined by:

x	y	$\bar{x}$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Satisfiability II

A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example

$f(x, y, z) = (x \wedge (y \vee \bar{z})) \vee (\bar{y} \wedge z \wedge \bar{x})$.

x	y	z	$(x \wedge (y \vee \bar{z}))$	$(\bar{y} \wedge z \wedge \bar{x})$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

For example, the assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.

Satisfiability III

Example

$$f(x, y) = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y}).$$

x	y	$x \vee y$	$\bar{x} \vee y$	$x \vee \bar{y}$	$\bar{x} \vee \bar{y}$	$f(x, y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x, y)$ true, and hence it is NOT satisfiable.

SAT $\in$ **NP**

SAT problem

Determine whether an input Boolean formula is satisfiable. If a Boolean formula is satisfiable, it is a yes-input; otherwise, it is a no-input.

Claim

$SAT \in \mathbf{NP}$.

Proof.

- The certificate consists of a particular 0 or 1 assignment to the variables.
- Given this assignment, we can evaluate the formula of length n (counting variables, operations, and parentheses), it requires at most n evaluations, each taking constant time.
- Hence, to check a certificate takes time $O(n)$.
- So we have SAT $\in$ **NP**.

□

k-SAT $\in$ **NP**

For a fixed k, consider Boolean formulas in k-conjunctive normal form (k-CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form

$$f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

Example (3-CNF formula)

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4).$$

k-SAT problem

Determine whether an input Boolean k-CNF formula is satisfiable.

Theorem

3-SAT $\in$ **NP**. *2-SAT* $\in$ **P**

Some Decision Problems in **NP**

We have given proofs for:

- Decision subset sum problem (DSubsetSum),
- Decision Hamiltonian cycle (DHamCyc),
- Satisfiability (SAT),
- Decision vertex cover problem (DVC).

Some others (without proofs given; try to find proofs):

- Decision minimum spanning tree problem (DMST),
- Decision 0-1 knapsack problem (DKnapsack).

$P = NP$?

One of the most important problems in computer science is whether $P = NP$ or $P \neq NP$?
Observe that $P \subseteq NP$.

- Given a problem $\pi \in P$, and a certificate, to verify the validity of a yes-input (an instance of π), we can simply solve π in polynomial time (since $\pi \in P$). It implies $\pi \in NP$.

Intuitively, $NP \subseteq P$ is doubtful.

- After all, just being able to verify a certificate (corresponding to a yes-input) in polynomial time does not necessary mean we can tell whether an input is a yes-input or a no-input in polynomial time.
- However, 45 years after the $P = NP$? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.