COMS W3261 : Complexity review

William Pires

1 Time Complexity

Say we have a TM M that decides a language L. Given some input x, we want to understand how long it takes M to accept or reject x. To do so, we consider the running time of M as a function of the length of x (how many alphabet symbols it takes to write x as input).

Definition 1. Let M be a Turing machine that halts on every input. The time complexity of M denoted $t(n)$ is the maximum number of steps M takes on any input of length n. That is :

$$t(n) := \max_{x \in \Sigma^*, |x|=n} \text{number of steps } M \text{ takes before it halts on } x$$

In the above, the number of **steps** of M on a input x is the number of transitions M takes before it halts on x.

Counting exactly how many steps a TM takes an input is quite cumbersome, so we use Big O notation to hide away constants.

Definition 2. Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f(n) = O(g(n))$ if : there exists constants c, n_0 such that :

$$\forall n \geq n_0 \;\; f(n) \leq c * g(n)$$

This means that once n is large enough, $f(n)$ is less than $g(n)$, up to constant factors. Here are some examples :

1. $n = O(n^2)$
2. $3n^2 + \log(n) = O(n^2)$
3. For any $a \in \mathbb{N}$, we have $\log_a(n) = O(\log_2(n))$.
4. $0.1 * n^3$ isn't $O(n^2)$. (Why ? Fix some constant c, then $0.1 * n^3$ will eventually always be larger than $c * n^2$ if we pick n to be large enough).

You should know that when you have a polynomial $f := a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 x + a_0$, we have $f(n) = O(n^k)$. This also means $a * n^k = O(n^k)$.

Definition 3 (Time($t(n)$))**.** Let $t : \mathbb{N} \to \mathbb{N}$, we define the class Time($t(n)$) as the following set of languages :

$$\text{Time}(t(n)) := \{ \; L \mid L \text{ is a language decided by an } O(t(n)) \text{ time Turing machine } \}$$

Unpacking the definition, this means there is some constant c and n_0, such that for all $n \geq n_0$, M halts in time $\leq c * t(n)$ on all inputs of length n and M decides L.

Definition 4. We say $t : \mathbb{N} \to \mathbb{N}$ is polynomial if there exists some constant k such that $f(n) = O(n^k)$.

Examples : $n^5 + 1000$, $log(n)$ are both polynomial. $2^n + n^3$ isn't polynomial.

2 P and NP

Definition 5 (P)**.**

$$\text{P} = \{\ L \mid L \text{ is a language and there exists a decider } M \text{ for } L \text{ running polynomial time } \}$$

Equivalently,

$$\text{P} := \bigcup_{k \in N} \text{Time}(n^k)$$

Proof Template 1 (Show $L \in$ P)**.**

1. Give pseudo-code of a deterministic decider for L
2. Show that if $x \in L$, your algorithm accepts.
3. Show that if $x \notin L$, your algorithm rejects.
4. Show that M runs in time $O(n^k)$ for some constant k.

A non-deterministic Turing machine is a Turing machine that is allowed to take non-deterministic transitions. That is given the current state q and the symbols the heads of the TM see on the different tapes, there could be multiple transitions possible. For instance, with a one tape TM with states Q and alphabet Σ, the transition function is now of the form $\sigma : Q \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{L, R\})$.

Definition 6. Let M be a non-deterministic TM. On input x, M starts in the initial configuration. And at each step, M can take any of the transitions possible based on it's current state and what the heads are seeing on the tapes.

- We say that M accepts x in time t, if there exists a sequence of t steps such that after these M accepts x.
- We say that M rejects x in time t, if for all sequences of t steps, M rejects x within these t steps.

In particular, for M to run in time $t(n)$, after at most $t(n)$ transitions, M must have halted on any input of length n. And this is no mater what transitions we non deterministically picked at each step.

We often think of such a TM as being able to "take a guess" and then checking if this guess is a solution. Here's an example :

Example 1. Consider the problem 3-coloring. In this problem you are given as input as input a graph G. G is given by a list V of n nodes, and a list E of m edges. The goal is to know if there exists a 3-coloring of G. That if for each node we can assign one of the colors $\{r, b, g\}$ and if (u, v) is an edge in G then u and v must have different colors.

This can be solves by a Non-deterministic TM as follow. Using non-determinism M can assign to each node a color from {r,b,g}. This gives us a coloring C_{guess}. [1]

Then, M goes over the list of m edges, and for each edge (u, v) it checks that u and v have different colors in C_{guess}. If M detects an edge (u, v) with the same color : it rejects. If after looking at all the edges M saw no violations, it accepts.

As in the determnistic case, we have the following two definitions :

Definition 7 (NTime($t(n)$))**.** Let $t : \mathbb{N} \to \mathbb{N}$, we define the class NTime($t(n)$) as the following set of languages :

NTime($t(n)$) := { L | L is a language decided by an $O(t(n))$ time **non-deterministic** Turing machine }

And the class NP is defined as :

Definition 8 (NP)**.**

NP = { L | L is a language decided by a polynomial time **non-deterministic** Turing machine }

Equivalently,

$$\text{NP} := \bigcup_{k \in N} \text{NTime}(n^k)$$

For problems in NP, it's often hard to deterministically see if $x \in L$ unless you use a brute force algorithm, but if $x \in L$, I can give you a "proof" y such that by looking at y you can be convinced $x \in L$ (in polynomial time).

This motivates an alternative definition for NP based on verifiers :

Definition 9 (Verifier)**.** A verifier V for a language L, is a **deterministic** algorithm such that V takes as as input x and some string c and

$$x \in L \leftrightarrow \exists\, c \text{ such that } V(x, c) \text{ accepts.}$$

[1] This can be done as follow. Imagine the input is on the first tape, M looks at the list of node in G one by one. Each time we read a node u, we have three possible transitions, which makes us write one of r, b, g on the second tape. This corresponds to the color we assign to u.

V is said to be a polytime verifier if it runs in time $O(|x|^k)$ for some constant k.

In the above we can think of c as the proof that $x \in L$. And if V is a polytime verifier we must always have $|c| = O(|x|^k)$, that is the proof must have polynomial length in $|x|$ (otherwise, the TM wouldn't even be able to read c in time $O(|x|^k)$) .

Example 2. Going back to the 3-Coloring example, a verifier for the problem is the following :

Algorithm 1 3-Coloring verifier

Input: $((V, E), C)$

▷ Here V a list of n nodes. E is a list of edges between the nodes in V. C is an assignment of $\{r, g, b\}$ to each node in V

```
for each edge (u, v) in E do
    if u, v have the same colors in C then
        Reject.                                  ▷ C isn't a valid coloring
    end if
end for
Accept.
```

The above works, because it runs in time $\mathcal{O}(|E| * |V|)$, since we look at all the edges in the graph, and each time check the color of the nodes on the edge (Everytime, we need to go through C to find the colors of u and v. And this takes time $O(|V|)$). So this runs in polynomial time in $|E|, |V|$.

As you saw in class we have the following equivalent definition of NP :

Theorem 1. NP = { L | $L$is a language and there exists a polytime verifier for L }

Thus, when asked to show $L \in$ NP, you have two options, give a polytime verifier for L or a polytime TM for L.

Proof Template 2 (Show $L \in$ NP using a Verifier)**.**

1. Give pseudo-code of a deterministic verifier V for L. Here V takes as input x and c (the certificate)
2. Show that if $x \in L$, there is some c such that V accepts (x, c) [a]
3. Show that if $x \notin L$, for all c, V rejects (x, c).
4. Show that V runs in time $O(|x|^k)$ for some constant k.

[a] It must be that $|c| = O(|x|^{k'})$ for some constant k'.

Proof Template 3 (Show $L \in$ NP using a non-deterministic TM)**.**

1. Give pseudo-code of a non-deterministic decider M for L. Here M takes as input x only.
2. Show that if $x \in L$, there is some non-deterministic choices that makes M accept.
3. Show that if $x \notin L$, M always reject x, no matter what the non-deterministic steps are.
4. Show that M runs in time $O(|x|^k)$ for some constant k.

Since a deterministic TM is a special case of a non-determinisitc TM we have the following :

Theorem 2. P $\subseteq$ NP

A major open question is whether NP $\subseteq$ P.

3 NP Completeness

There are many problems in NP and for the ones that aren't in P, we don't know any polynomial time algorithm to solve them. This motivates the notion of NP completeness and NP hardness.

Definition 10 (Poly-time mapping reducible)**.** A language $A \subseteq \Sigma^*$ is polynomial-time (mapping) reducible to $B \subseteq \Sigma^*$ (written A $\leq_{\text{P}}$ B) if there is a polynomial-time function $f : \Sigma^* \to \Sigma^*$ such that :

$$x \in A \iff f(x) \in B$$

In particular, we must have $|f(x)|$ is polynomial in $|x|$.

In particular this means the following, say I had a polynomial time algorithm M for B. Then I get get a polynomial time algorithm M' for A as follow :

Algorithm 2 A polytime algorithm for A given a polytime decider M for B

Input: (x)

Compute $f(x)$ ▷ This takes polynomial time since f is poly-time computable

Run M on $f(x)$ ▷ M runs in polynomial time in $|x|$
▷ Since $|f(x)|$ is polynomial in $|x|$, running M takes polynomial time in $|x|$

if M accepts $f(x)$ **then**
  Accept x
else if M rejects $f(x)$ **then**
  Reject x
end if
▷ This clearly runs in polynomial time in $|x|$

This in turn motivates the following definition :

Definition 11 (NP-Hardness)**.** A language B is NP hard, if **for all** languages $A \in$ NP we have that there is a polynomial time reduction from A to B (A $\leq_{\mathrm{P}}$ B).

In particular, if we were to find a polynomial time for an NP-hard problem, we would have that NP = P. Another important definition is that of NP completeness :

Definition 12 (NP-Complete)**.** A language B is NP complete iff $B \in$ NP **and** B is NP hard.

Obviously, you might wonder, how we can show a problem is NP hard in the first place. How would we be able to show that **all** languages in NP are polytime reducible to some language ? This is the seminal result of Cook and Levin who independently proved the following :

Theorem 3. 3-SAT is NP Complete.

When asked to prove a language L is NP hard or NP complete, you should proceed as follow :

Proof Template 4 (Show that A is NP Hard)**.**

1. Pick NP-hard problem B.
2. Give pseudo-code for a function f.
3. Show that given x as input $f(x)$ can be computed in time $O(|x|^k)$ for some constant k.
4. Show that if $x \in A$, then $f(x) \in B$
5. Show that if $x \notin A$, then $f(x) \notin B$

.

It might not be clear why A is NP-hard, according to the definition of NP-Hardness, when we you follow the above proof template. So here's the details of why this is correct.

We picked NP-hard problem B and showed $B \leq_{\mathrm{P}} A$. So let L be any language in NP, by definition of NP-Hardness we have $L \leq_{\mathrm{P}} B$. Thus, there exists a polynomial time computable function g such that $x \in L \iff g(x) \in L$. And in the template we showed there exists a polynomial time computable function f such that $y \in B \iff f(y) \in A$.

So we claim $f \circ g$ is a polynomial time function and that $x \in L \iff f(g(x)) \in A$. Assuming these two claims are true, we have $L \leq_P A$ (by definition), and thus any language $L \in \mathrm{NP}$ is polynomial time reducible to A, so A is $\mathrm{NP} - hard$.

First, let's show this is computable in polynomial time. Since g is polynomial time computable, given x as input, we can compute $g(x)$ in polynomial time. So we know $|g(x)|$ is polynomial in $|x|$ (otherwise, we wouldn't even have the time to write $g(x)$ in polynomial time). Since f is also computable in polynomial time, we can compute $f(g(x))$ in polynomial time in $|g(x)|$. But this is also polynomial in $|x|$ (since $|g(x)|$ is polynomial in $|x|$). So this shows that $f \circ g$ is computable in polynomial time.

Now we want to show this $x \in L \iff f(g(x)) \in A$.

- For the first direction, if $x \in L$, then we must have $g(x) \in B$. But we know $y \in B \Rightarrow f(y) \in A$, so $g(x) \in B \Rightarrow f(g(x)) \in A$, so putting it all together, $x \in L \Rightarrow g(x) \in B \Rightarrow f(g(x)) \in A$.
- For the other direction, assume $f(g(x)) \in A$, then it must be that $g(x) \in B$ by the property of f. But also, by property of g, if $g(x) \in B$ we must have $x \in L$. So $f(g(x)) \in A \Rightarrow x \in L$.

So we proved both direction, so the claim is true.

Proof Template 5 (Show that A is NP Complete)**.**

1. Show $A \in$ NP
2. Show A is NP hard

Here's the list of NP of NP-complete problems to use in your reductions :

- 3-SAT. Given a CNF formula where every clause has 3 literals, is the formula satisfiable ?
- Hampath. Given a graph G and two nodes s, t is there an Hamiltonian path from s to t in G ?
- Clique. Given a graph G and an integer k, does G contain a k-clique (G contains as a subgraph the complete graph on k vertices) ?

4 Good to know : How to deal with subsets

Often a problem will require to look at all the subsets of size k of some set. If you have a set with n elements, the number of subsets of size k is denoted $\binom{n}{k}$. **It's important to know that :**

$$\binom{n}{k} = O(n^k)$$

Here's an example. The problem 3-Sum is defined as follow. You are given as input a set S of n numbers in $\mathbb{Z}$. The goal is to know whether there is any way to pick 3 distinct element a, b, c in S such that $a + b + c = 0$.

To solve this problem, it's enough to look at all the subsets of 3 elements in S and check if the 3 elements sum to 0. Thus we need to look at $\binom{n}{3}$ subsets of S and for each subset, we check if $a + b + c = 0$ (you can assume this check takes constant times). So the running time is $O\left(\binom{n}{k}\right) = O(n^3)$, which is polynomial.

5 A note about k-SAT

This section is here to help you understand what a k-CNF is, since it plays such a big role in this part of the course.

When dealing with a boolean formula we have **variables** $x_1, \ldots, x_n$. An assignment to the variables $x_1, \ldots, x_n$ means we give each x_i value 0 (false) or the value 1 (true). In particular, if we have n variables there is 2^n possible assignments.

A **literal** is a boolean variable x or it's negation $\bar{x}$ (I.e. if we assign $x = 0$ then $\bar{x} = 1$ and if $x = 1$ then $\bar{x} = 0$).

A clause is a $\vee$ (logical OR) of literals. For example $C_1 := x_1$, $C_2 := \bar{x}_1 \vee x_3 \vee \bar{x}_4$ are example of clauses.

Given an assignment of 0/1 to each variable x_i, we say the assignment satisfies a clause C, if **at least one literal** of C evaluate to 1.

Consider the clause C_2 above. To satisfy C_2 we need at least one of $\bar{x}_1, x_3$ or $\bar{x}_4$ to be set to 1. Thus is we assign $x_1 = 0$ or $x_3 = 1$ or $x_4 = 0$ then C_2 is satisfied. An assignment falsifies C_2 if it sets $x_1 = 1, x_3 = 0$ and $x_4 = 1$, since then all the literals of C_2 evaluate to 0.

A CNF formula ϕ is the $\wedge$ of a bunch of clauses. Here are some examples :

- $\phi := (x_1) \wedge (x_2 \vee \bar{x}_3)$
- $\phi := (x_1) \wedge (x_2 \vee \bar{x}_4 \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_4)$

What does it to satisfy a CNF formula ϕ ? If ϕ contains variables $x_1, \ldots, x_n$, then we must assign each x_i to 0 or 1 so that **all** the clauses of ϕ are satisfied. Sometimes, it's impossible to satisfy a CNF formula.

Definition 13. A k-CNF formula, is a CNF formula where all clauses have at most k literals

Here's an example of a 3-CNF :

$$\phi := (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x_2} \vee \bar{x_3} \vee \bar{x}_4) \wedge (x_1 \vee x_2 \vee x_5)$$

A k CNF formula has at most $\binom{2n}{1} + \binom{2n}{2} + \ldots + \binom{2n}{k}$ clauses, since for each clause we pick at most k literals from $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$.

To check a k-CNF formula ϕ is satisfiable, we can use a brute force algorithm that goes over all possible assignments to the variables. In particular, we have n variables, where each can take 0/1 value. There is 1 assignment will all variables set to 0, $\binom{n}{1}$ assignments with 1 variable set to 1, $\binom{n}{2}$ assignments with 2 variable set to 1, etc... So in total there is :

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \text{ many possible assignments}$$

Checking an assignment satisfies ϕ means checking that each clause has at least one literal evaluating to 1, so that takes time $O(|\phi|)$. So the run time of this brute force algorithm is $O(2^n * |\phi|)$.[2]

Here's an example of the brute force algorithm on a CNF. We go over all assignments to the variables, and for each clause check if it's satisfied. If they are all satisfied then ϕ is satisfied :

Example 3.

$$\phi := (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1) \wedge (\bar{x}_4 \vee \bar{x}_2) \wedge x_4$$

x_1	x_2	x_3	x_4	$x_1 \vee x_2$	$x_2 \vee x_3$	$\bar{x}_3 \vee \bar{x}_1$	$\bar{x}_4 \vee \bar{x}_2$	x_4	ϕ
0	0	0	0	0	0	1	1	0	0
0	0	0	1	0	0	1	1	1	0
0	0	1	0	0	1	1	1	0	0
0	0	1	1	0	1	1	1	1	0
0	1	0	0	1	1	1	1	0	0
0	1	0	1	1	1	1	0	1	0
0	1	1	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1	0	0
1	0	0	1	1	0	1	1	1	0
1	1	0	0	1	1	1	1	0	0
1	0	1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1	1	0
1	1	0	1	1	1	1	0	1	0
1	1	1	0	1	1	0	1	0	0
1	1	1	1	1	1	0	0	1	0

You can see that for all truth assignment, we always have one clause that evaluates to 0 (no littera is set to true), thus ϕ isn't satisfiable.

[2]This is exponential when $|\phi|$ is polynomial is n. This is always the case for a k-CNF where k is constant.

6 Exercises

Exercise 1 (True False)**.** 1. All of $n * \log(n)$, $n^100 + 3n^2$, $n^{0.001n}$, $2^{\sqrt{n}}$ are polynomial.

2. The runtime of a verifier $V(x, c)$ must be polynomial in $|x| + |c|$.
3. All NP Complete problems are polytime reducible to each other.
4. Say the input of a TM is an integer n given in binary, and M runs in time $O(n^2)$, then M runs in polynomial time.

Exercise 2. Show that NP is closed under unions. That is if $L_1, L_2 \in$ NP then $L_1 \cup L_2 \in$ NP.

Exercise 3.

- Fix some constant k. The problem k-Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. You can think of G as being described by a $n \times n$ binary matrix A where $A_{i,j} = 1$ iff there's an edge (i, j) in G (so the input size is n^2). You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why is this problem in P ?
- The NP complete problem Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. **But now k is given to you as input in decimal**. (So the input size is roughly $(n^2 + \log_{10}(k))$). Again, You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why doesn't the previous proof work to show this problem is in P ?

Exercise 4. Problem 7.18 in the book. Show that if P = NP,then every language $A \in$ P, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete. (Hint : think about the definition of $L \leq_{\mathrm{P}} A$, knowing that $L \in$ NP implies $L \in$ P.)

Exercise 5. Problem 7.21 in the book. Let G represent an undirected graph. Also let

- SPATH = $\{(G, a, b, k)$— G contains a simple path of and length **at most** k from a to $b\}$
- LPATH = $\{(G, a, b, k)$— G contains a simple path of and length **at least** k from a to $b\}$

Show that SPATH $\in$ P and LPATH is NP-complete. (A simple path is a path that doesn't visit the same node twice.)

7 Solutions

Exercise 1 (True False)**.**

1. All of $n * \log(n)$, $n^{100} + 3n^2$, $n^{0.001n}$, $2^{\sqrt{n}}$ are polynomial.
   Answer : $n * \log(n)$, $n^{100} + 3n^2$ are polynomial, $n^{0.001n}$, $2^{\sqrt{n}}$ aren't.
2. The runtime of a verifier $V(x, c)$ must be polynomial in $|x| + |c|$.
   Answer : False, the runtime can only depend on $|x|$. [3]
3. All NP Complete problems are polytime reducible to each other.
   Answer : True. Here's a proof.
   Let A, B be NP-complete, we want to show $A \leq_{\mathrm{P}} B$.
   We know B is NP complete, and thus it's NP hard. So for any $L \in \mathrm{NP}$, we have $L \leq_{\mathrm{P}} B$.
   Since A is NP complete, it means $A \in \mathrm{NP}$.
   Combining the two, we have $A \leq_{\mathrm{P}} B$.
4. Say the input of a TM is an integer n given in binary, and M runs in time $O(n^2)$, then M runs in polynomial time.
   Answer : False. n is given as input in binary, so the input size is $O(\log_2(n))$. But $n^2 = 2^{2\log_2(n)}$, using $n = 2^{\log_2(n)}$. So the runtime is exponential in the input length.

Exercise 2. Show that NP is closed under unions. That is if $L_1, L_2 \in \mathrm{NP}$ then $L_1 \cup L_2 \in \mathrm{NP}$.

Let $L_1, L_2 \in \mathrm{NP}$. Then by definition, there exists a polynomial time verifier V_1, such that :

$$x \in L_1 \Leftrightarrow \exists c_1 \text{ such that } V_1(x, c_1) \text{ accepts.}$$

Similarly, we have a polynomial time verifier V_2 for L_2 such that :

$$x \in L_2 \Leftrightarrow \exists c_2 \text{ such that } V_2(x, c_2) \text{ accepts.}$$

We now want to build a polynomial time verifier V for $L_1 \cup L_2$. The idea is that given c, we first run V_1 to check if it accepts, and then we run V_2. We accept if either accepts, and reject if they both reject.

[3] But this means that if the verifier V accepts (x, c), then it must be that the length of c is polynomial in $|x|$. Otherwise, V wouldn't have the time to read c in polynomial time.

Algorithm 3 A verifier for $L_1 \cup L_2$

Input: (x, c)

```
Run V_1 on x, c                          ▷ V_1 runs in polynomial time in |x|
if V_1 accepts then
    Accept x
end if

Run V_2 on x, c                          ▷ V_2 runs in polynomial time in |x|
if V_2 accepts then
    Accept x
end if
Reject.
```

First, it's clear that V runs in polynomial time in $|x|$, as all it does is run V_1, V_2 on x, c. [4]

So it remains to show

$$x \in L_1 \cup L_2 \iff \exists\, c \text{ such that } V(x, c) \text{ accepts.}$$

Assume $x \in L_1 \cup L_2$, we want to show there exists a c such that $V(x, c)$ accepts. If $x \in L_1$, we know there exists a c_1 such that $V_1(x, c_1)$ accepts. So taking $c = c_1$ leads to V accepting.
Else if $x \in L_2$, we know there exists a c_2 such that $V_2(x, c_2)$ accepts. So taking $c = c_2$ leads to V accepting.

Thus

$$x \in L_1 \cup L_2 \Rightarrow \exists\, c \text{ such that } V(x, c) \text{ accepts}$$

We now show the other direction. Assume there exists a c such that $V(x, c)$ accepts. Then in the pseudocode either $V_1(x, c)$ or $V_2(x, c)$ must have accepted. If V_1 accepted, then by definition

$$V_1(x, c) \text{ accepts } \Rightarrow x \in L_1$$

Else if V_2 accepted then by definition

$$V_2(x, c) \text{ accepts } \Rightarrow x \in L_2$$

Anyway, it must be that $x \in L_1 \cup L_2$. So $V(x, c)$ accepts $\Rightarrow x \in L_1 \cup L_2$.

Exercise 3.

- Fix some constant k. The problem k-Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. You can think of G as being described by a $n \times n$ binary matrix A where $A_{i,j} = 1$ iff there's an edge (i, j) in G (so the input size is n^2). You must

[4] Here's a technical point you can ignore. We should first check that c isn't too large. By that, I mean $|c| = O(n^k)$ where we have V_1, V_2 both run in time $O(n^k)$. I.e, if c is bigger than the runtime of V_1 and V_2, we should reject. This to avoid the case where c is so large that giving it as input to V_1 would take too long. And if c is bigger than the runtime of V_1, V_2 we know they would never accept (x, c).

accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why is this problem in P ?

We can give a polynomial time algorithm for the problem as follow :

Algorithm 4 An algorithm for k-Clique

Input: (G)

for all subset S of k vertices from G **do**
Check that for each pairs (u, v), with $u, v \in S$ we have that (u, v) is an edge in G.
▷ This means the vertices in S form a complete graph.
If for all pairs, the edge (u, v) is in G, accept.
end for
Reject.

This algorithm basically looks at all the subsets of k vertices in G and checks they form a complete graph. Clearly if $G \in$ k-Clique, then we will accept, and otherwise we will rejects.

In the algorithm, we look at all the $\binom{n}{k}$ subsets of k vertices of G. For each subset, we need to check $O(k^2)$ edges are present, one for all pair. Since k is a constant, we can assume this takes constant time. So the runtime of the algorithm is $O(n^k)$. Since k is a constant, this is polynomial time.

- The NP complete problem Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. **But now k is given to you as input in decimal**. (So the input size is roughly $(n^2 + \log_{10}(k))$). Again, You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why doesn't the previous proof work to show this problem is in P ?

  If we had to adapt the above algorithm for this proof it would look like that :

Algorithm 5 An algorithm for Clique

Input: (G, k) ▷ k is now part of the input

for all subset S of k vertices from G **do**
Check that for each pairs $(u, v) \in S$ (u, v) is an edge in G.

if for all pairs, the edge (u, v) is in G **then**
Accept.
end if

end for

Reject.

Imagine the input is $\left(G, k = \frac{n}{2}\right)$. Then, the algorithm will have to look at $\binom{n}{\frac{n}{2}}$ subsets, but that is roughly 2^n many subsets. However, the input size is only $n^2 + \log_{10}(n)$, so the runtime of the

algorithm is exponential. [5]

Exercise 4. Problem 7.18 in the book. Show that if P = NP,then every language $A \in$ P, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete. (Hint : think about the definition of $L \leq_P A$, knowing that $L \in$ NP implies $L \in$ P.)

This exercise is great to test how comfortable you feel with the definition of NP completeness.

Assume P = NP and let $A \in$ P, be any language except $\emptyset$ or Σ^*. That means there exists two strings y, z such that $y \in A$ and $z \notin A$.

Now consider the NP complete problem 3-SAT, since P = NP by assumption, we have 3-SAT $\in P$. So there exists a polynomial time decider M for 3-SAT.

We want to show A is NP-hard. We will show 3-SAT $\leq_P A$. So, we claim the following is a mapping reduction from 3-SAT to A.

Algorithm 6 A mapping reduction f from 3-SAT to A

Input: (ϕ) ▷ ϕ is a CNF

Run M on ϕ ▷ This takes polynomial time
▷ This tells us if $\phi \in$ 3-SAT or not

if M accepts **then**
Return y ▷ $\phi \in$ 3-SAT, so we return $y \in A$

else
Return z ▷ $\phi \notin$ 3-SAT, so we return $z \notin A$

end if

It's clear the above mapping f is computed in polynomial time. Besides it's also clear that if $\phi \in$ 3-SAT, we have $f(\phi) = y \in A$. And if $\phi \notin$ 3-SAT, then $f(\phi) = z \notin A$. So $f(\phi) \in A \iff \phi \in$ 3-SAT.

So this is a valid polynomial time mapping reduction from 3-SAT to A.

So A is NP-hard, and since $A \in$ P, $A \in$ NP and thus by definition A is NP complete.

You might wonder why we need that A isn't $\emptyset$ or Σ^*. In a mapping reduction from a language L to A , we need $x \in L \Leftrightarrow f(x) \in A$.

But if we let $A = \emptyset$, then there's no strings in A ! So if $x \in L$, we can never have $f(x) \in A$. So there can't be a mapping reduction from L to A if $L \neq \emptyset$.

[5].It's important to realize we look at the running time in the WORST CASE input. And here, the algorithm is very slow whenever you give $k = n/2$ (or any large enough function of n).

Similarly, if $A = \Sigma^*$, then all strings are in A. So if $x \notin L$, we can never have $f(x) \notin A$. So there can't be a mapping reduction from L to A if $L \neq \Sigma^*$.

Exercise 6. Problem 7.21 in the book. Let G represent an undirected graph. Also let

- SPATH = $\{(G, a, b, k)$— G contains a simple path of and length **at most** k from a to $b\}$
- LPATH = $\{(G, a, b, k)$— G contains a simple path of and length **at least** k from a to $b\}$

Show that SPATH $\in$ P and LPATH is NP-complete. (A simple path is a path that doesn't visit the same node twice.)

Recall that a simple path is a path that never visits a node twice (i.e. the path has no cycles).

We first show SPATH $\in P$. To check there is a simple path of length $\leq k$ from a to b it suffices to do the following : Perform a Breath-First Search starting at a in G.

When we perform a BFS, we first see all the nodes at distance 1 from a, then we see the ones at distance 2, etc... So if we reach b before we're at distance $k+1$, we accept, else we reject. Obviously doing a BFS takes times polynomial in the size of the input.

We now show LPATH is NP-Complete.

First, we show LPATH is in NP. Here's a verifier for LPATH :

Algorithm 7 A verifier for LPATH

Input: $((G, a, b, k), p)$ ▷ p is the extra string the verifier takes as input

Check p is a simple path from a to b in G.
Check that p has length at least k.

if Both of the above are true **then**
    Accept.
else
    Reject.
end if

Clearly, the above runs in polynomial time. Also it's clear that if $(G, a, b, k) \in$ LPATH, then there must be a simple path of length $\geq k$ between a and b. So just set p to be this path, this will lead the verifier to accept. Obviously, if no such path exists, there's no p that would lead the verifier to accept $(G, a, k, b), p$.

So it remains to show the problem is NP-hard. To do this we will reduce the NP-Complete problem HamPath to LPATH. We need to give a polynomial time mapping reduction from Hampath to LPATH.

Recall that HamPath is the following problem : Given a graph G and two nodes s, t is there an Hamiltonian path from s to t in G.

Given G, s, t as input for Hampath, our function outputs $(G, a = s, b = t, k = n)$. I.e. $f(G, s, t) = G, s, t, n$.

Clearly this is computable in polynomial time. So it remains to show that $(G, s, t) \in$ Hampath if and only if $(G, s, t, n) \in$ LPATH.

This isn't hard, once we unpack the definition of Hamiltonian path. By definition there is an Hamiltonian path from s to t, if and only if, there is a simple path between s and t that visits every vertex once.

Since the path starts at s, ends at t and goes through every of the n vertices once, it means it must be of length n [6].

So by definition G has an Hamiltonian path from s to t if and only if there is a simple path between s and t of length n.

Also note that simple paths can't have length more than n, else a vertex would be repeated so it wouldn't be simple.

Thus, we clearly have that there's an Hamiltonian path path from s to t if and only if there is a simple path between s and t of length at least n.

So $G, s, t \in$ HamPath iff $G, s, t, n \in$ LPATH. This proves Hampath $\leq_{\mathrm{P}}$ LPATH. So LPATH is NP-hard.

Since LPATH is NP-hard and in NP, it's NP complete.

[6] Here I count the length of a path as number of vertices