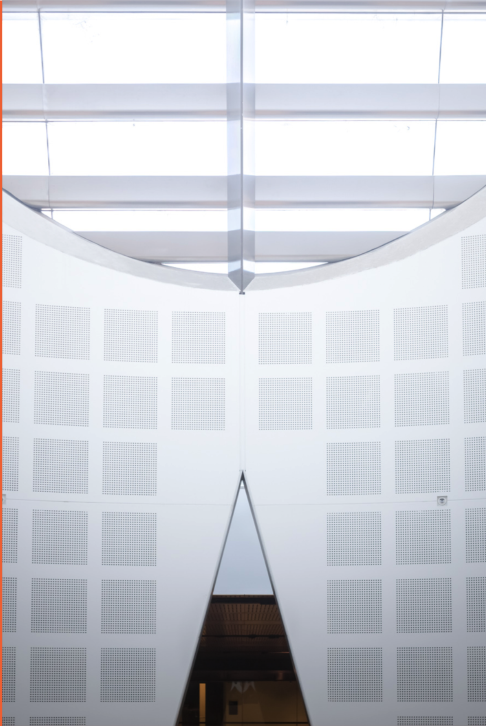
COMP2022
Models of Computation
Intro to Computational Complexity
P vs NP
Sipser Chapter 7

Sri AravindaKrishnan Thyagarajan (Aravin)

September 19, 2024

THE UNIVERSITY OF SYDNEY



Agenda

Investigate the time (= number of steps) taken by TMs to solve problems.

- What is time complexity?
- What is the **P** vs **NP** problem?

Recall that a polynomial (with integer coefficients) in the variable n is an expression of the form

$$\sum_{i=0}^{k} a_i n^i$$

where each a_i is an integer and $k \in \mathbb{Z}_0^+$.

e.g., $2n^3 - 3n + 1$ is a polynomial.

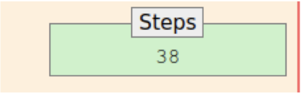
What is a step?

What counts as a step for pseudocode?[1]

- assignments $a = 42$
- comparisons $i < j$
- Boolean formulas $(p\&q)\&q$
- mathematical operations $a = 42x + y$

What counts as a step for Turing machines?

- A single transition counts as a step.
- The number of steps of a run on a given input is computed by the simulator http://morphett.info/turing/



[1]This is how we do it in COMP2x23.

What is time complexity?

Me: How many steps does an algorithm/TM take?

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

Me: Right. Let's restrict to deciders. What if we look at inputs of a particular length, say, n?

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

Me: Right. Let's restrict to deciders. What if we look at inputs of a particular length, say, n?

You: Sure, but strings of the same length may take a different number of steps.

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

Me: Right. Let's restrict to deciders. What if we look at inputs of a particular length, say, n?

You: Sure, but strings of the same length may take a different number of steps.

Me: Right. So let's look at the worst case input.

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

Me: Right. Let's restrict to deciders. What if we look at inputs of a particular length, say, n?

You: Sure, but strings of the same length may take a different number of steps.

Me: Right. So let's look at the worst case input.

You: So for each length n we look at the largest number of steps amongst all inputs of length n.

What is time complexity?

Me: How many steps does an algorithm/TM take?

You: It depends on the input. Different inputs can take different number of steps. And on some inputs they may run forever.

Me: Right. Let's restrict to deciders. What if we look at inputs of a particular length, say, n?

You: Sure, but strings of the same length may take a different number of steps.

Me: Right. So let's look at the worst case input.

You: So for each length n we look at the largest number of steps amongst all inputs of length n.

Me: Right! This is a function $f(n)$, called the time complexity of the algorithm/TM

$f(n) =$ the largest number of steps taken by the algorithm/TM on any input of length n.

We say that the algorithm/TM runs in time $f(n)$.

If $f(n) = O(p(n))$ for some polynomial p, then we say that the algorithm/TM runs in polynomial time

For instance, M has time complexity $n \log n$ then M runs in polynomial time (since $n \log n = O(n^2)$).

```
1 def my_fun(array A of integers):
2   n = size(A)
3   for i = 1 to n-1:
4     for j = i+1 to n:
5       if A[i] > A[j] then swap A[i] with A[j]
6   return A
```

- On arrays of size n, line 5 is executed $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$ steps.
- So, the time complexity of this algorithm is $\Theta(n^2)$, i.e., quadratic.[2]
- We call it a *quadratic-time algorithm*

[2] Recall that a function f is $\Theta(g)$, read "Big-Theta g", if there are constants c, d, e such that $cg(n) \leq f(n) \leq dg(n)$ for all $n > e$.

This TM M decides the language $L = \{0^n 1^n : n \geq 1\}$

The time complexity of M is $\Theta(n^2)$ (Tutorial).

The most important class of languages

Definition

Define **P** to be the collection of languages L that are decidable in polynomial time on **deterministic** TMs.

- Read "P" or "P Time" or "Poly Time" or "Polynomial Time" or "Deterministic Polynomial Time".
- **P** includes all the languages decided by linear-time algorithms, and quadratic-time algorithms, and cubic-time algorithms, etc.
- **P** is robust to certain changes in the model, notably multiple tapes.

P roughly corresponds to the problems that can be realistically solved on a computer.

Examples of languages in P

- Every regular language. Why?
- Every context-free language. Why?
- Some of the problems studied in this course, *e.g.*, DFA membership, RE membership, DFA equivalence, CFG membership, CFG emptiness
- Most of the problems studied in
  - `COMP2123:Data Structures and Algorithms`
  - `COMP3027:Algorithm Design`

Self-test

The halting-problem is not in **P** because...

1. It is recognisable, and some recognisable languages are in **P**.
2. It is not decidable, but every language in **P** is decidable.
3. Actually, it is in **P**, and the statement above is wrong!

We saw that automata had deterministic and nondeterministic versions. Turing machines do too!

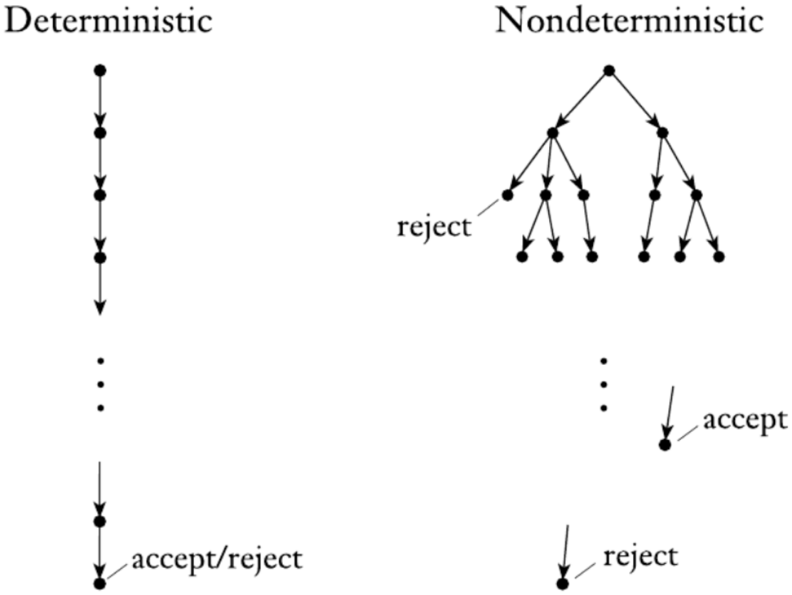
Non-Deterministic TMs

The type of the transition function of a non-deterministic TM N becomes:

$$\delta : Q \times \Gamma \to P(\Gamma \times \{L, R, S\} \times Q)$$

E.g., $(a', L, q') \in \delta(q, a)$ means "if N is in state q and reads symbol a under the head, one of its possible transitions is to write a', change to state q', and move the head one cell to the left".

A computation of N on an input u is a tree, called the computation tree


Deterministic
Nondeterministic
reject
accept
accept/reject
reject

What does it mean for a NTM to accept an input?

Definition

A NTM N accepts input u if some branch of the computation tree of u has an accepting configuration.

So, if no branch of the tree has an accepting configuration (because each rejects or diverges) then N does not accept u.

Theorem

Every non-deterministic TM N has an equivalent deterministic TM D.

The idea is that D will "search the computation tree of N".

High-level description of TM D:

- D does a breadth-first search of N's computation tree on given input.
- If it finds q_{accept} it accepts, otherwise it diverges.

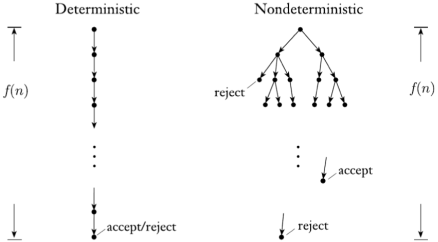
Suppose D were to use a **depth-first search (DFS)** instead. Would this work?

1. Yes, because we know that BFS and DFS both end up traversing the whole tree.
2. No: D might reach an accepting configuration even if N can't.
3. No: D might reach a rejecting configuration even if N can't.
4. No: D might not reach an accepting configuration even if N can.
5. No: D might not reach a rejecting configuration even if N can.

What is time complexity for NTMs?

Definition

1. An NTM N is a decider if on every input, every branch of its computation tree halts.
2. The time-complexity of N is the the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.
3. If $f(n) = O(p(n))$ for some polynomial p then N runs in polynomial time



The second most important class of languages

Definition
Define **NP** to be the collection of languages L that are decidable in polynomial time on **nondeterministic** Turing machines.

– read "NP" or "Nondeterministic Polynomial time"

– It does not stand for "Not Polynomial"

What languages are in **NP**?

– all languages that are in **P**. Why?

What other languages are in **NP**?

Graphs (AK)

A graph G is a pair (V, E) where V is a set of *vertices* and $E \subseteq V \times V$ is a set of *edges*.

Example:

- A vertex represents a city
- An edge represents a two-way road between two cities

A non-empty graph is called a clique (aka "completely connected") if every pair of different nodes is connected by an edge.

CLIQUE is in NP

The CLIQUE problem:
Input: Graph (V, E) and $K \in \mathbb{Z}^+$
Output: "Yes" if the graph contains K vertices that form a clique, and "No" otherwise.

To show that CLIQUE is in **NP**, we only need to give a polynomial time NTM that decides it.

Here is a high-level description of such a TM:

On input $(V, E), K$:

1. Nondeterministically select a subset $C \subseteq V$ of K vertices.
2. Test whether every pair of different nodes in C is connected by an edge.
3. If yes, accept; else reject

The most(?) important unsolved problem in computer science

We know that every problem in **P** is also in **NP**, i.e., $\mathbf{P} \subseteq \mathbf{NP}$

However, we don't know if these are equal:

Is **P** equal to **NP**?

You: Ok, so we know that every language in **P** is also in **NP**, but we don't know the reverse.

Me: Right. Do you think CLIQUE is in **P**?

You: Ok, so we know that every language in **P** is also in **NP**, but we don't know the reverse.

Me: Right. Do you think CLIQUE is in **P**?

You: I can't think of a faster way than checking all sets of vertices of size K, and there are about $|V|^K$ such sets, which is exponential in K

Me: Right. In fact, if we could show that CLIQUE was in **P** we would be able to conclude that $\mathbf{P} = \mathbf{NP}$.

You: Ok, so we know that every language in **P** is also in **NP**, but we don't know the reverse.

Me: Right. Do you think CLIQUE is in **P**?

You: I can't think of a faster way than checking all sets of vertices of size K, and there are about $|V|^K$ such sets, which is exponential in K

Me: Right. In fact, if we could show that CLIQUE was in **P** we would be able to conclude that $\mathbf{P} = \mathbf{NP}$.

You: (mind-blown emoji)

Me: And there are lots of problems like this. They are called **NP**-complete, and are in some sense the "hardest" problems in **NP**.

You: Ok, so we know that every language in **P** is also in **NP**, but we don't know the reverse.

Me: Right. Do you think CLIQUE is in **P**?

You: I can't think of a faster way than checking all sets of vertices of size K, and there are about $|V|^K$ such sets, which is exponential in K

Me: Right. In fact, if we could show that CLIQUE was in **P** we would be able to conclude that $\mathbf{P} = \mathbf{NP}$.

You: (mind-blown emoji)

Me: And there are lots of problems like this. They are called **NP**-complete, and are in some sense the "hardest" problems in **NP**.

You: Where can I learn more about **NP**-completeness?

Me: `COMP3027:Algorithm Design`