Lecture 19

Universal Turing Machines

Turing machines give a general model of computation. But, so far, every Turing machine we have met has been able to solve one specific problem: recognise one specific language or compute one specific function. For every new task, we designed a completely new Turing machine which served as a special-purpose computational device, or program, for that specific task.

But a computer is a much more general tool. It can be programmed to do any computable task. We will see that computers can also be modelled by a type of Turing machine called a Universal Turing Machine.

19.1 Encoding Turing machines

In the previous lecture, we discussed how to encode some kinds of objects — including numbers and sequences — as strings, so that they can be given as input to Turing machines.

We now consider how to encode *Turing machines themselves* as strings, so that they can be given as input to other Turing machines!

There are many ways to do this. We give one particular coding scheme, making many arbitrary choices on the way.

We assume our Turing machine has the following attributes.

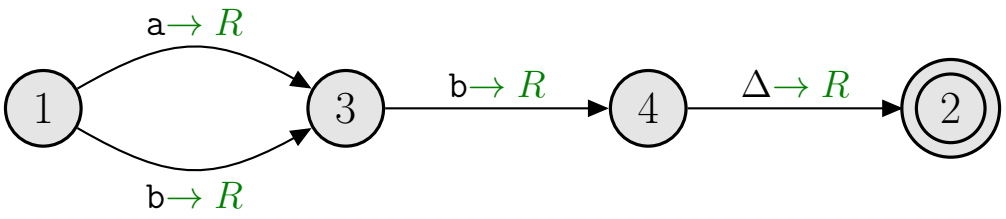
- Input Alphabet: {a,b}
- Tape Alphabet: {a,b,#}
- Start State: numbered 1
- Accept State: numbered 2

So far, we have usually represented Turing machines as diagrams. But we mentioned that they can be represented as tables, and it is the tabular representation that lends itself to being encoded as a string, by converting each row of the table into a string and then concatenating all the rows. We need to do this in such a way that all the details of the Turing machine can be recovered from the long string we use to encode it.

We will represent a Turing machine as a table in which each row specifies one transition, with the entries in the row giving, in order: the state the transition goes out *from*; the state

the transition goes *to*; the letter that this transition must *read*; the letter that this transition must *write*; the direction in which the tape head *moves*.

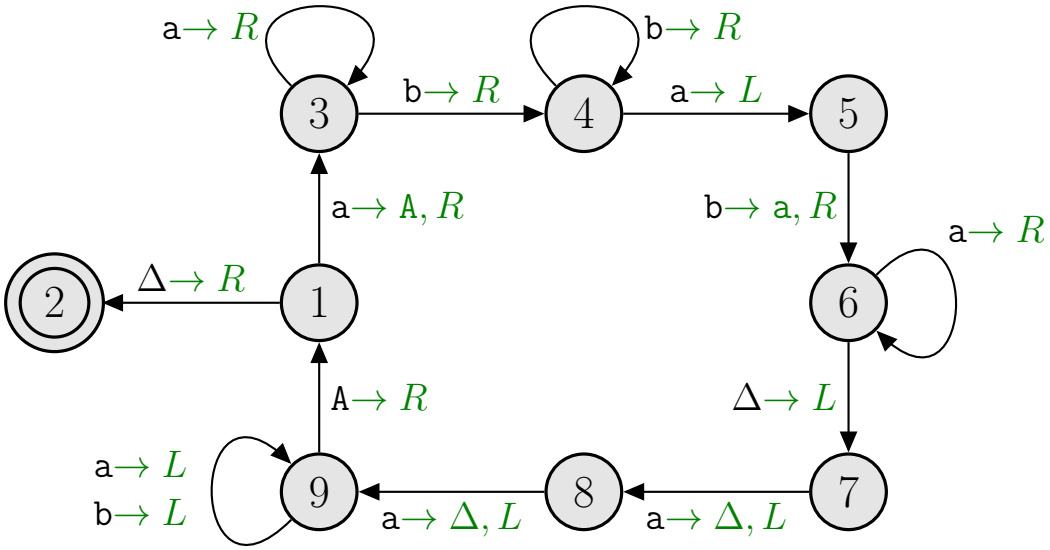
Here is a Turing machine for the two-string language {ab, bb}.



Here it is as a table.

From	To	Read	Write	Move
1	3	a	a	R
1	3	b	b	R
3	4	b	b	R
4	2	Δ	Δ	R

As another example, here is a Turing machine for the non-context-free language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n : n \geq 0\}$, from Lecture 18.



Here is its table.

From	To	Read	Write	Move
1	3	a	#	R
3	3	a	a	R
3	4	b	b	R
4	4	b	b	R
4	5	a	a	L
5	6	b	a	R
6	6	a	a	R
6	7	Δ	Δ	L
7	8	a	Δ	L
8	9	a	Δ	L
9	9	a	a	L
9	9	b	b	L
9	1	#	#	R
1	2	Δ	Δ	R

In order for a table to represent a valid Turing machine, some conditions must be checked.

- There is a row with a 1 in the From column. (Start state)
- There is no row with a 2 in the From column. (No transitions out of the Accept state)
- There are no two rows with the same numbers in the From and the same letter in the Read column. (Determinism)

To encode the table as a string, we need to encode each entry in the table — numbers for states, input alphabet letters, tape alphabet letters, symbols representing directions — as a string over our alphabet {a,b}. Here is the scheme we will use.

State number	Code
n	$\mathtt{a}^n\mathtt{b}$

Letter	Code
a	aa
b	ab
Δ	ba
#	bb

Direction	Code
L	a
R	b

Using this scheme, the rows of our second Turing machine above can be encoded as follows.

From	To	Read	Write	Move	Code
1	3	a	#	R	abaaabaabbb
3	3	a	a	R	aaabaaabaaaab
3	4	b	b	R	aaabaaaababab
4	4	b	b	R	aaabaaaababab
4	5	a	a	L	aaaabaaaaabaaaaa
5	6	b	a	R	aaaaabaaaaaababaab
6	6	a	a	R	aaaaaabaaaaaabaaaab
6	7	Δ	Δ	L	aaaaaabaaaaaaabbabaa
7	8	a	Δ	L	aaaaaaabaaaaaaaabaabaa
8	9	a	Δ	L	aaaaaaaabaaaaaaaaabaabaa
9	9	a	a	L	aaaaaaaaabaaaaaaaaabaaaaa
9	9	b	b	L	aaaaaaaaabaaaaaaaaabababa
9	1	#	#	R	aaaaaaaaabaabbbbbb
1	2	Δ	Δ	R	abaabbabab

Concatenating all these row-strings gives one long string that encodes the entire Turing machine:

```
abaaabaabbbaaabaaabaaaabaaabaaaababababbaaabaaaababababbaaaabaaaaabaaaaa
  aaaaabaaaaaababaabaaaaaabaaaaaabaaaabaaaaaabaaaaaaabbabaa
  aaaaaaabaaaaaaaabaabaaaaaaaaaabaaaaaaaaabaabaaaaaaaaaaabaaaaaaaaabaaaaa
  aaaaaaaaabaaaaaaaaababababaaaaaaaaaabababbbbbbbabaabbabab
```

. . . as one long string, without line breaks or spaces.

You should satisfy yourself that this encoding is *reversible*: we can recover the entire Turing machine from it, without any loss of information.

To discuss strings constructed using our encoding scheme, it is helpful to introduce the following language.

The **Code-Word Language (CWL)** is the regular language defined by the regular expression

$$(\mathtt{aa}^*\mathtt{baa}^*\mathtt{b}(\mathtt{a}\cup\mathtt{b})^5)^*$$

Words which encode Turing machines belong to CWL.
BUT not all words in CWL encode a Turing machine.

Let's use this last sentence to get some more practice in working with quantifiers. The sentence says:

$$\neg\,\forall w \in \text{CWL} \quad \exists M \;:\; w \text{ encodes } M$$

Starting with this sentence, and moving the negation from left to right and making the necessary changes as we go, we obtain the following sequence of sentences.

$$\begin{aligned}
\neg\,\forall w \in \text{CWL} \quad &\exists M \;:\; w \text{ encodes } M \\
\exists w \in \text{CWL} \quad \neg\,&\exists M \;:\; w \text{ encodes } M \\
\exists w \in \text{CWL} \quad &\forall M \;:\; \neg\,(w \text{ encodes } M) \\
\exists w \in \text{CWL} \quad &\forall M \;:\; w \text{ does not encode } M
\end{aligned}$$

Our final sentence has the same logical meaning as the original. It says that there is a word in the Code Word Language that does not encode any Turing machine.

To see how to decode strings in CWL that represent Turing machines, try decoding the string

`abaaabaaaababaaababababbaaabaaaababababbaaaabaabbabab`

We obtain the following table, which represents the four-state Turing machine we presented earlier.

From	To	Read	Write	Move
1	3	a	a	R
1	3	b	b	R
3	4	b	b	R
4	2	Δ	Δ	R

Here is an outline of the decoding algorithm.

While there are unread letters

1. Read and count the next clump of a's, then read the b after it.
   - Interpret clump of a's as the state number, in unary, that this transition goes from.
2. Read and count the next clump of a's, then read the b after it.
   - Interpret clump of a's as the state number, in unary, that this transition goes to.
3. Read the next two letters.
   - Interpret it as the letter to be read for this transition.
4. Read the next two letters.
   - Interpret it as the letter to be written for this transition.
5. Read the next letter.
   - Interpret it as the direction for this transition.

Now that we can encode Turing machines as strings, we can give them as inputs to other Turing machines. This enables us to use Turing machines to do computations about other Turing machines. For example, we could build a Turing machine M which, given an encoding of another Turing machine P as its input, could compute the number of states of P. Or it could count the transitions of P, or find how many transitions go into the Accept State of P, or determine if there are any loop transitions in P, and so on. In fact, we can do a great deal more than that.

19.2 Universal Turing Machine (UTM)

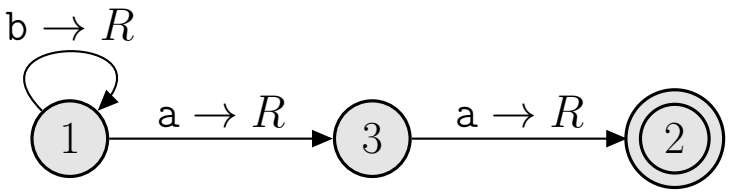
A **Universal Turing Machine (UTM)** is a Turing Machine that takes, as input,

- an encoding of some Turing Machine M, together with
- a string x, to be used as input to M

and simulates the execution of M on x.

What does the input to a Universal Turing Machine look like? First, there is an encoding of some Turing machine M, then some separator (we'll use `$`) to mark the end of that part of the input, then the string x which we want M to run on.

Suppose we have the following Turing machine.



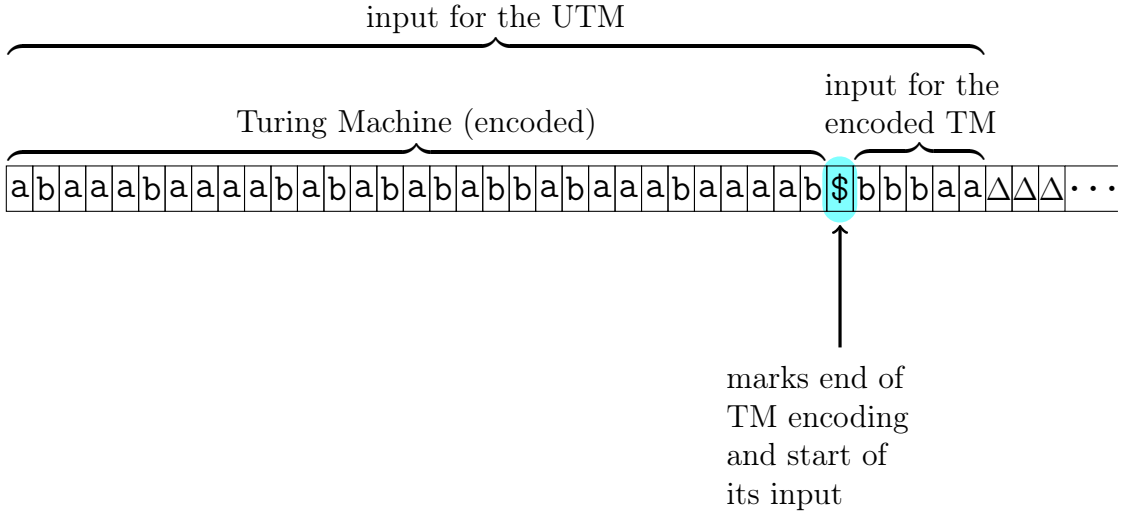
Using our encoding procedure, this TM can be encoded as the string

`abaaabaaaababababbaaabaabaaaab`

Suppose we want a UTM to simulate the execution of this TM on the string `bbbaa`. Then the input to the UTM consists of two parts:

Turing Machine:	`abaaabaaaababababbaaabaabaaaab`
Data:	`bbbaa`

On the UTM's tape, it looks like this:



The process by which a UTM simulates the operation of its input TM on an input string is summarised in the following algorithm.

1. Move rightwards to first letter of the encoded TM's input.
   Read it. Mark it, so we can come back to it (e.g., $\mathtt{a} \mapsto \mathtt{A}$, $\mathtt{b} \mapsto \mathtt{B}$).
   - Remember it (in choice of state).
2. Move leftwards to first instruction.
3. If the next state in current instruction in encoded TM is the Accept state

   Find (from current instruction) what to write and direction of next move.
   Remember it (in choice of state).

   Move rightwards back to current position in encoded TM's input.

   Write the required letter, move in the required direction, and Accept.

   else

   Find (from current instruction) what to write and direction of next move.
   Remember it (in choice of state).

   Move rightwards back to current position in encoded TM's input.

   Write the required letter, move in the required direction.

   Read current letter in encoded TM's input. Mark it, so we can come back to it.

   Remember it (in choice of state).

   Move leftwards to find the next instruction (using remembered letter).

So far, we have merely defined UTMs, described how their input can be laid out on the tape, and outlined an algorithm for one of them. We have not given an example of a UTM or proved that one exists. But, having outlined an algorithm for a UTM, we can be confident that one exists, by the Church-Turing thesis. And, in fact, UTMs have been written, some with remarkably few states.

Exercise

Suppose:

- U is a UTM,
- T is a TM
- x is an input string for T, with $|x| = n$.
- When T is run on input x, it takes time t and visits at most s tape cells.

Using the algorithm outline of the previous slide, and the encoding scheme for TMs given in this lecture:

- Determine an upper bound for the time taken by U to simulate the running of T on input x.

- Give the bound in terms of t, s and n.

19.3 UTMs and stored-program computers

Universal Turing Machines are important because they give us a model of one computer simulating another. They also give a formal model of a **stored-program** computer, where the program to be executed is given to the computer as data (in effect, a string) and stored in the computer's working memory just as the program's input data is. Most computers today are of the stored-program type; in fact, the stored-program characteristic is often taken to be part of the *definition* of what it means to be a computer.

It is a good challenge to design your own Universal Turing Machine. Once you do so, you can truly be said to have *designed your own computer*. Although doing this will take significant time and effort, there should be a great sense of achievement once it is done. You will have completed a rite of passage into the top tier of young computer scientists.

Lecture 20

Decidability

Recall from Lecture 18 that a *decider* is a Turing Machine that halts for every input, and a language L is *decidable* if it is Accept(M) for some decider M (in which case, its complement is Reject(L) for the same decider).

Some synonyms for decidable include **recursive** and **solvable**. The term "computable" is also sometimes used as a synonym for decidable, but that can be confusing as it is also used to with a different meaning.

In this lecture, we give several important examples of decidable languages and discuss some properties of the class of decidable languages.

We have already seen that all regular languages are decidable, and indeed all context-free languages are decidable. There are certainly some non-context-free languages that are decidable, for example $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n : n \geq 0\}$.

20.1 Decision Problems

When describing languages, we have mainly used standard set notation. This is natural because languages are just sets (of strings). But now is a good time to introduce the *Decision Problems*, which are another convenient way to describe languages.

A **decision problem** consists of a statement of the type of *input* that is to be considered, followed by a *question* about that input which always has a Yes/No answer.

A decider **solves** a decision problem if it

- Accepts an input for which the answer is Yes, and
- Rejects any input for which the answer is No.

Compare this definition with the definition of a *decider* from Lecture 18. This comparison will indicate the close relationship between languages and decision problems.

Here are some examples of decision problems about a variety of different types of objects. For each of them, think about how the inputs could be encoded as strings, and what sort of techniques you might use to solve them.

INPUT: an integer
QUESTION: Is it even?

INPUT: a string.
QUESTION: Is it a palindrome?

INPUT: an expression in propositional logic
QUESTION: Is it ever True?

INPUT: a graph G, and two vertices s and t
QUESTION: is there a path from s to t in G ?

INPUT: a Python program
QUESTION: is it syntactically correct?

Here are some more examples, focusing on some of the main concepts we have met in this unit.

INPUT: a Finite Automaton
QUESTION: Does it define the empty language?

INPUT: two Regular Expressions
QUESTION: Do they define the same language?

INPUT: a Finite Automaton
QUESTION: Does it define an infinite language?

INPUT: a Context Free Grammar
QUESTION: Does it define the empty language?

INPUT: a Context Free Grammar
QUESTION: Does it generate an infinite language?

INPUT: a Context Free Grammar and a string w
QUESTION: Can w be generated by the grammar?

Every *decision problem* defines an associated *language*: just take the set of all YES-inputs for the decision problem. (To treat these YES-inputs as the members of a language, there has to be some way of encoding the inputs as strings. This is always possible for the examples we consider.)

Conversely, every *language* defines an associated *decision problem*, namely:

INPUT: a string (over some alphabet, usually representing some object)
QUESTION: Is the string in the Language?

Thus, a decider solves a decision problem if and only if it is a decider for its corresponding language.

Since the input and output for a Turing Machine is always a string, other objects need to be encoded as strings in order to be used by Turing machines.

For any object O, $\langle O \rangle$ will denote encoding of the object as a string.

If we have several objects, $O_1, \ldots, O_n$, we denote their encoding into a single string by $\langle O_1, \ldots, O_n \rangle$.

20.2 Testing Emptiness of Regular Languages

Decision Problem:

INPUT: a Finite Automaton
QUESTION: Does it define the empty language?

Language:

$$\text{FA-Empty} := \{\langle A \rangle : A \text{ is a FA and } L(A) = \emptyset\}$$

Theorem 35
FA-Empty is decidable.

Algorithm:

Input: $\langle A \rangle$ where A is a Finite Automaton.

1. Mark the Start State of A.
2. Repeat until no new states get marked:
   - Mark any state that has a transition coming into it from any state that is already marked.
   - If no final state is marked, Accept; otherwise Reject.

20.3 Testing Equivalence of Regular Expressions

Decision Problem:

REGULAR EXPRESSION EQUIVALENCE
INPUT: two Regular Expressions
QUESTION: Do they define the same language?

For a Regular expression R, let $L(R)$ be the language defined by R.

Language:

$$\text{RegExpEquiv} := \{\langle A, B \rangle : A, B \text{ are regular expressions and } L(A) = L(B)\}$$

Theorem 36
RegExpEquiv is decidable.

Algorithm:

Input: $\langle A, B \rangle$ where A and B are regular expressions

1. Construct a FA, C, that defines the language

$$(L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)).$$

2. Run our decider for FA-Empty on C.

3. If that decider accepts C, then Accept, else Reject.

20.4 Testing Emptiness of Context Free Language

Decision Problem:

INPUT: a Context Free Grammar
QUESTION: Does it define the empty language?

Language:

$$\text{CFG-Empty} := \{G : G \text{ is a CFG and } G \text{ defines the empty language}\}$$

Theorem 37
CFG-Empty is decidable.

Algorithm:

Input: $\langle A \rangle$ where A is a Context Free Grammar.

1. Mark all the terminal symbols in A.

2. Repeat until no new symbols get marked:

   - Mark any non-terminal X that has a production which has all the right-hand symbols marked.
   - If Start Symbol is not marked, Accept, else Reject.

It is worth comparing this algorithm with the Nullability algorithm from Lecture 16, although keep in mind that the problems the two algorithms solve are different.

20.5 Some Decidable Problems

Here is a list of some decidable problems. It includes the problems we have considered in this lecture, some problems we have solved in previous lectures, and some that are good exercises.

INPUT: a Finite Automaton
QUESTION: Does it define the empty language?

INPUT: two Regular Expressions
QUESTION: Do they define the same language?

INPUT: a Finite Automaton
QUESTION: Does it define an infinite language?

INPUT: a Context Free Grammar
QUESTION: Does it define the empty language?

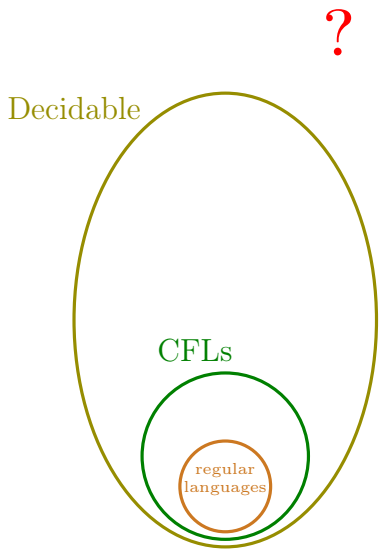
INPUT: a Context Free Grammar
QUESTION: Does it generate an infinite language?

INPUT: a Context Free Grammar and a string w
QUESTION: Can w be generated by the grammar?

20.6 Language classes

Our world of language classes now looks like the following Venn diagram.

The class of decidable languages is in fact much bigger than the class of context-free languages. But does it contain *everything*?



20.7 Closure properties

As usual, when we meet a new language class, we consider its closure properties. So it is time to do this for the class of decidable languages.

Theorem 38
If L is decidable, then $\overline{L}$ is decidable.

Theorem 39
If L_1 and L_2 are decidable, then so are

- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $L_1 L_2$

Exercise:
Formulate and prove more closure results.

Revision

Reading: Sipser, Section 4.1, pp. 190–201.

Preparation: Sipser, Section 4.2, pp. 201–213, especially pp. 207–209.

DECIDABLE

① checking a number is even

Decidable

Problem statement: given integer x, is x even?

Decidability proof: if $n\%2 == 0$ → TRUE (is even)
else → FALSE (not even)

since the algorithm is deterministic and can solve for an input given and provide correct ans, its decidable.

② membership in finite language

Decidable

Problem statement: given finite language (↗ a set of strings), determine if string x is a member of that language

Decidability proof:
- finite language = finite list of strings in language
- check if x is in the finite list of strings
  - ↳ TRUE → a member
  - ↳ FALSE → not a member

③ regular expression matching

Decidable

Problem statement: given a regular expression, decide string x match the regular expression

Decidability proof: regular expression ⇒ pattern matching using NFA or DFA:

```
import re
re.match(regex, string x)   # regex ↗ regular expression
```

↳ TRUE → match
↳ FALSE → x match

④ sorting a list of string/number

- string → lexicographically order
  - ↳ sort using sorting algorithm (eg. merge sort)
  - ↳ Decidable
- number → ... 1 < 2 < ... < 1000
  - ↳ sort using sorted() function
  - ↳ Decidable
- a list of number + string (eg. [1, 2, five, six])
  - ↳ sort using sorted() function
    - ↳ sorted(mixed-list, key=lambda x: (isinstance(x, int), x))
  - ↳ Decidable

↳ integers sorted before non-int

sort string lexicographically

Lecture 21

Mapping Reductions

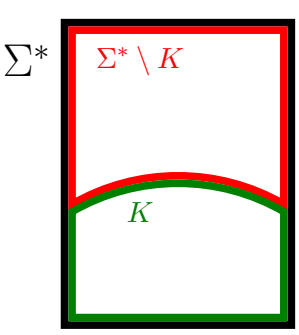
Our emphasis so far has been on making decisions for a single language: specifically, deciding whether or not a given string is in the language. We now consider computational relationships *between* languages.

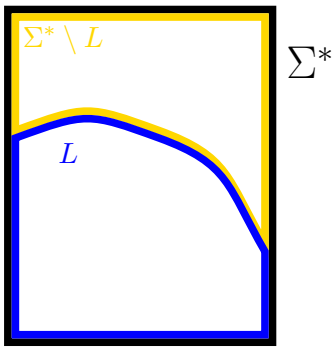
21.1 Mapping reductions

We have been learning about deciders, which are theoretical models of computational solutions to decision problems (see Lectures 18 and 20). But we often don't solve problems in isolation: one common problem-solving strategy is to reduce the problem we *want* to solve to another problem that we *already know how* to solve. Furthermore, even if we can't relate our problem to an already-solved problem, it can still be helpful to relate one problem to another, computationally. Such a relationship can enable us to use knowledge we gain about one problem to shed light on another problem.

The most fundamental way of relating one language to another is by *mapping reductions*.

Suppose you have two languages, K and language L, over our alphabet Σ^*. Here they are, in separate Venn diagrams.



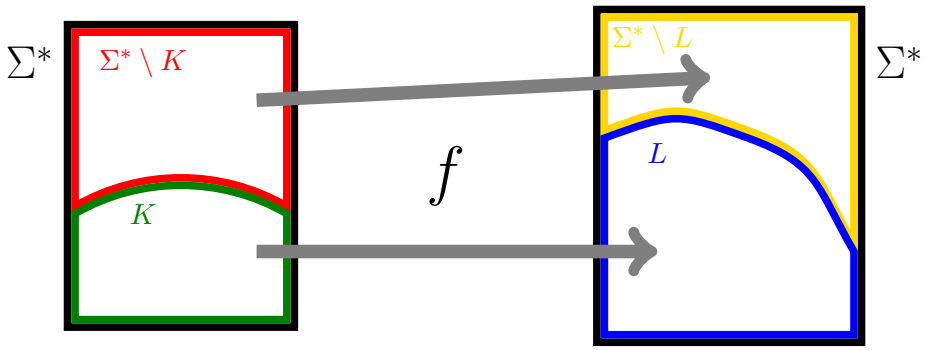


A **mapping reduction** from language K to language L is a computable function $f : \Sigma^* \to \Sigma^*$ such that, for every $x \in \Sigma^*$,

$$x \in K \qquad \text{if and only if} \qquad f(x) \in L.$$

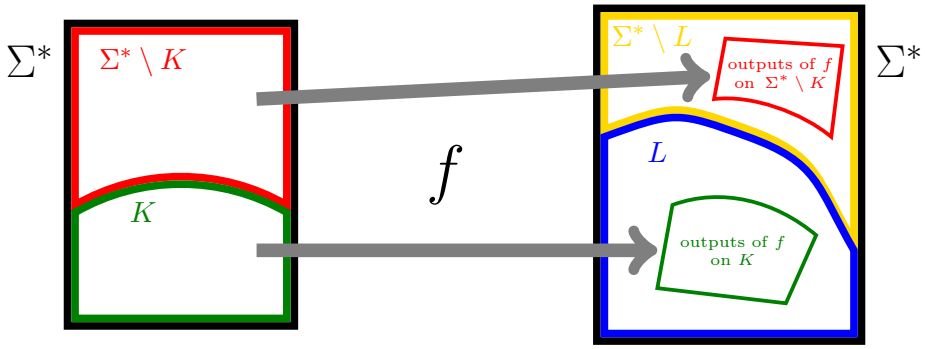
Notation: we write $K \leq_m L$ to mean that *there exists* a mapping reduction from K to L.

Using our Venn diagrams, we can represent the action of f by the long arrows. They indicate that f sends members of K to members of L and non-members of K to non-members of L.



So the outputs of f applied to K form a subset of L. Similarly, the outputs of f when applied to $\Sigma^* \setminus K$ form a subset of $\Sigma^* \setminus L$.

These subsets are usually *proper* subsets: the outputs of f when applied to K usually give a *proper* subset of L, and the outputs of f when applied to $\Sigma^* \setminus K$ give a *proper* subset of $\Sigma^* \setminus L$. This is illustrated in the next diagram.



Mapping reductions must treat strings in K differently to strings outside K, because the former are mapped into L and the latter are mapped to outside L. BUT this does *not* mean that the reduction needs to test if a string belongs to K or not, in order to decide what to do with it. In fact, the *essence* of mapping reductions is to ensure that the membership condition — $\forall x : x \in K \Leftrightarrow f(x) \in L$ — is satisfied, *without* testing for membership of K or L. We will return to this crucial point later.

Let's get one trivial observation out of the way quickly before proceeding. Every language is mapping-reducible to itself:

$$\forall L : L \leq_m L.$$

The mapping reduction f that does this is just the identity function, i.e., the function that does nothing, so its output always equals its input.

The importance of mapping reductions comes from the next two theorems.

Firstly, a mapping reduction from your problem to a decidable problem yields a decider for your problem.

Theorem 40
If there is a mapping reduction f from K to L, then:

If L is decidable, then K is decidable.

Symbolically:

$$(K \leq_m L) \wedge (L \text{ is decidable}) \implies (K \text{ is decidable})$$

Proof.
We show that the following algorithm is a decider for K.

Input: x.

Compute $f(x)$.

Run the Decider for L on $f(x)$.

// *This L-Decider accepts $f(x)$ if and only if $x \in K$, since f is a mapping reduction from K to L.*

This is indeed a computable process, by the computability of f (part of the definition of a mapping reduction) and the fact that we use a decider for L.

If $x \in K$ then $f(x) \in L$, since f is a mapping reduction. Therefore the L-Decider accepts $f(x)$, which makes the given algorithm accept its input x.

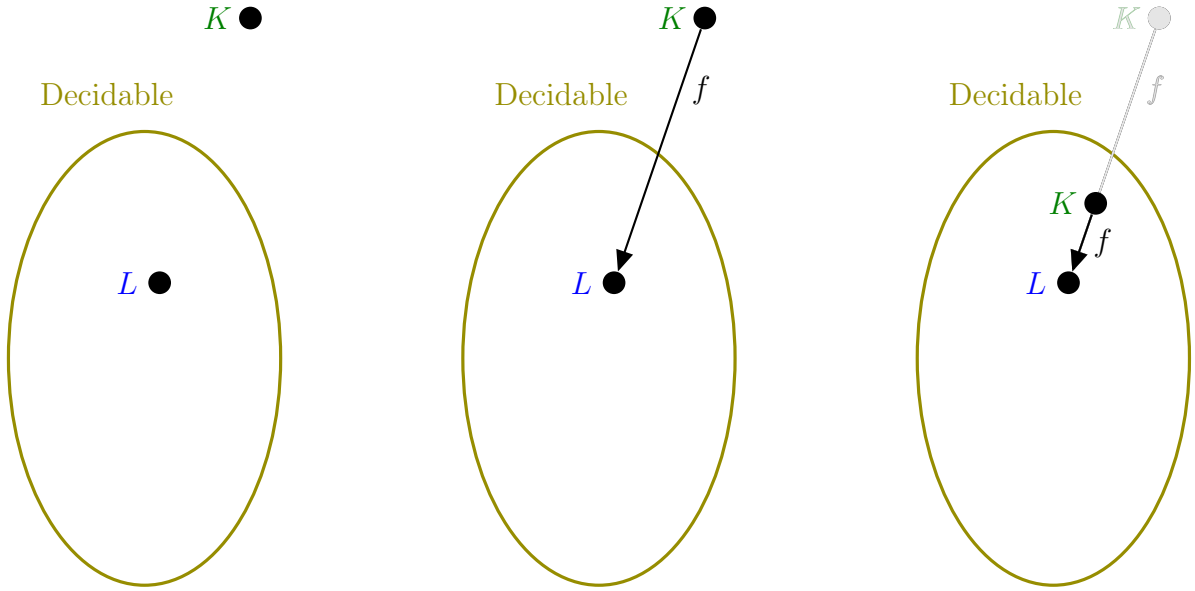
If $x \notin K$ then $f(x) \notin L$, since f is a mapping reduction. Therefore the L-Decider rejects $f(x)$, which makes the given algorithm reject its input x.

Therefore the given algorithm is a decider for K. □

So, suppose we have a language K which we don't yet know to be decidable. (See the left diagram below.)

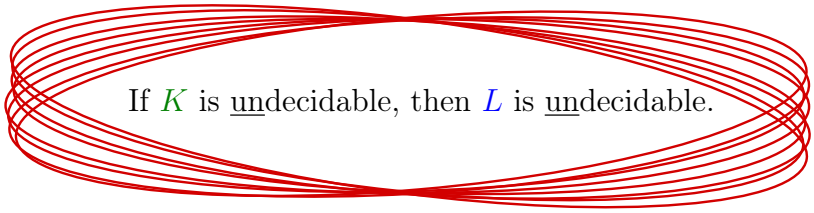
Suppose then that we find a mapping reduction from K to some language L that we already know to be decidable. Then we might try to visualise things as in the middle diagram.

But Theorem 40 tells us that the situation is actually as shown in the right diagram, with K now known to be in the class of decidable languages.



Conversely, a mapping reduction *from* an undecidable problem *to* your problem proves that your problem is undecidable.

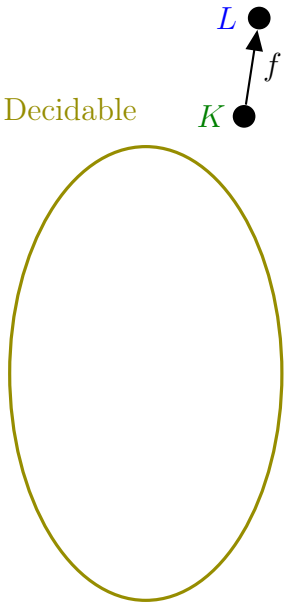
Corollary 41
If there is a mapping reduction f from K to L, then:



Symbolically:

$$(K \leq_m L) \wedge (K \text{ is undecidable}) \implies (L \text{ is undecidable})$$

Proof.
Contrapositive of previous Theorem. □



We have not yet shown that undecidable languages exist, let alone seen any examples of undecidable languages. So we are not yet in a position to use mapping reductions in the manner of Corollary 41. We shall use mapping reductions in that way in the next Lecture. For now, we use mapping reductions and Theorem 40, with some decidable languages we have met previously, to show that certain languages are decidable.

To prove that a function is indeed a mapping reduction from a language K to another language L you must prove that the two parts of the definition of a mapping reduction are satisfied: firstly, that the function is computable; secondly, that for any input x, we have $x \in K \Leftrightarrow f(x) \in L$.

We now give some examples of mapping reductions using languages we have met previously.

21.2 EQUAL to HALF-AND-HALF

Let f be the function computed by the following algorithm.

> Input: a word w over alphabet {a,b}
>
> Sort w
>
> Output the sorted word.

We consider f with respect to each of the two conditions that a mapping reduction must satisfy.

Firstly, f is computable because it only requires sorting.

Secondly, f satisfies the membership condition because

$$\begin{aligned} w \in \text{EQUAL} &\iff \text{it has the same number of } \texttt{a}\text{'s as } \texttt{b}\text{'s} \\ &\iff \text{after sorting, it has the same number of } \texttt{a}\text{'s as } \texttt{b}\text{'s} \\ &\qquad \text{(since sorting does not affect letter frequencies)} \\ &\iff f(w) \text{ consists of some number of } \texttt{a}\text{'s followed by} \\ &\qquad \textit{the same number} \text{ of } \texttt{b}\text{'s} \\ &\iff f(w) \in \text{HALF-AND-HALF}. \end{aligned}$$

So f satisfies the two requirements for a mapping reduction.

Observe also — and this is a key point — that the reduction ***does not test for membership of EQUAL***. It does the sorting *without knowing* whether or not the string being sorted has the same number of as and bs.

21.3 HALF-AND-HALF to PARENTHESES

A first draft at a mapping reduction from HALF-AND-HALF to PARENTHESES might look like this:

> Input: a word w
>
> For each letter of w in turn:
>
> > replace current letter as follows:
> > $\texttt{a} \mapsto \texttt{(}$
> > $\texttt{b} \mapsto \texttt{)}$
>
> Output: the string obtained from w by doing all these replacements.

This does much of what is required. Firstly, any string in HALF-AND-HALF gets mapped to a string that has some number of opening parentheses, $\texttt{(}\cdots\texttt{(}$, followed by *the same number* of closing parentheses, $\texttt{)}\cdots\texttt{)}$, which belongs to PARENTHESES. Secondly, any string where the numbers of as and bs are unequal (so it's not in HALF-AND-HALF) gets mapped to a string consisting of unequal numbers of parentheses of the two types, so the string produced does not belong to PARENTHESES, as required. But, at present, there are some strings *not* in HALF-AND-HALF that get mapped to something in PARENTHESES. For example, `abab` is not in HALF-AND-HALF, yet it is mapped to `()()` which does belong to PARENTHESES. So we need to detect strings that do not consist just of an `a`-stretch followed by a `b`-stretch, and treat them differently. With these considerations in mind, we revise our algorithm to produce the following.

> Input: a word w
>
> For each letter of w in turn:

If previous letter was `b` and current letter is `a`

// We have just seen `ba` which is impossible in HALF-AND-HALF.

Output the string `)`.

else

replace current letter as follows:

`a` $\mapsto$ `(`

`b` $\mapsto$ `)`

Output: the string obtained from w by doing all these replacements.

This function is computable because all its parts — looping over letters, detecting the substring `ba`, and replacement of letters by appropriate parentheses — are easy to compute. Above, we gave most of the reasoning for the rest of the proof that it is a mapping reduction.

21.4 EQUAL to PARENTHESES

Is there a mapping reduction from EQUAL to PARENTHESES? Yes! We can just compose the two previous mapping reductions.

This is a special case of:

Theorem 42

Mapping reducibility is transitive:

$$K \leq_m L \leq_m M \implies K \leq_m M.$$

Proof.

Let f be a mapping reduction from K to L, and let g be a mapping reduction from L to M.

We claim that the composition $g \circ f$, defined for all w by $g \circ f(w) = g(f(w))$, is a mapping reduction from K to M.

Since f and g are both computable, $g \circ f$ must be too.

Now we show that $g \circ f$ preserves membership in the required way.

$$\begin{aligned} w \in K &\iff f(w) \in L && \text{(since } f \text{ is a mapping reduction from } K \text{ to } L\text{)} \\ &\iff g(f(w)) \in M && \text{(since } g \text{ is a mapping reduction from } L \text{ to } M\text{)} \\ &\iff (g \circ f)(w) \in M && \text{(by definition of } g \circ f\text{)}. \end{aligned}$$

□

21.5 FA-Empty to No-Digraph-Path

Recall these definitions from Lecture 20:

$$\begin{aligned}
\text{FA-Empty} &:= \{\langle A\rangle : A \text{ is a FA and } L(A) = \emptyset\} \\
\text{Digraph-Path} &:= \{\langle G, s, t\rangle : G \text{ is a directed graph, } s, t \text{ are vertices in } G, \text{ and} \\
&\qquad \text{there exists a directed } s\text{–}t \text{ path in } G.\} \\
\text{No-Digraph-Path} &:= \{\langle G, s, t\rangle : G \text{ is a directed graph, } s, t \text{ are vertices in } G, \text{ and} \\
&\qquad \text{there does not exist a directed } s\text{–}t \text{ path in } G.\}
\end{aligned}$$

We give a mapping reduction from FA-Empty to No-Digraph-Path.

Input: $\langle A\rangle$ where A is a Finite Automaton.

1. Construct the directed graph G of A:
   - initially, vertices of G := states of A
   - every transition $v \xrightarrow{x} w$ in A becomes a directed edge (v, w) from v to w in G.
   - then add a new vertex t
   - for every Final State v of A, add a new directed edge (v, t) from v to t in G.
2. Specify s and t:
   - s := vertex of Start State of A.
   - t is as created above (the new vertex).
3. Output: $\langle G, s, t\rangle$

It is evident that the function is computable.
It also preserves membership as required:

$$\begin{aligned}
A \in \text{FA-Empty} &\iff \text{there is no sequence of transitions in } A \text{ leading} \\
&\qquad \text{from Start State to a Final State} \\
&\iff \text{there is no path in } G \text{ leading from } s \text{ to a vertex} \\
&\qquad \text{representing a Final State} \\
&\iff \text{there is no path in } G \text{ leading from } s \text{ to } t \\
&\iff \langle G, s, t\rangle \in \text{No-Digraph-Path}
\end{aligned}$$

Observe, again, that this mapping reduction does not determine whether or not $L(A) = \emptyset$. In effect, it transforms A into a digraph G (with vertices s and t) so that our original question, about emptiness of $L(A)$, becomes a question about nonexistence of a directed s–t path in G, with the answers to the two questions being the same.

21.6 RegExpEquiv to FA-Empty

Recall the following language from Lecture 20:

$$\text{RegExpEquiv} := \{\langle A, B\rangle : A, B \text{ are regular expressions and } L(A) = L(B)\}$$

We give a mapping reduction from RegExpEquiv to FA-Empty. In fact, we actually gave one in the previous lecture! But it was embedded in another algorithm. The decider we gave in §20.3 for RegExpEquiv just constructed an input for FA-Empty and then used the FA-Empty decider. The part of that algorithm that constructed the input for FA-Empty was nothing more or less than a mapping reduction.

We give the construction again, this time framing it as a mapping reduction.

Input: $\langle A, B\rangle$ where A and B are regular expressions

1. Construct a FA, C, that defines the language
$$(L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)).$$
2. Output: C

21.7 Reducing *from* a decidable language

We close with some technical comments giving some more basic properties of mapping reductions which also serve to identify some situations where mapping reductions serve no real purpose. Once you know these situations, you can avoid falling into the trap of using mapping reductions for them.

We have seen that mapping reductions are important for reducing to a decidable problem (in order to prove decidability of something else) or for reducing from and undecidable problem (in order to prove the undecidability of something else).

We can also ask about doing mapping reductions from a decidable problem.

To illustrate, let's reduce from EnglishPalindromes to YearsOfTransitsOfVenus, where the latter language is defined as follows.

$$\begin{aligned}\text{YearsOfTransitsOfVenus} &:= \{n : \text{a Transit of Venus occurs in year } n\ \} \\ &:= \{\ldots, 1761, 1769, 1874, 1882, 2004, 2012, 2117, \ldots\}\end{aligned}$$

Mapping reduction:

Input: a string w over the English alphabet

If w is a palindrome

output 2012

else

output 2023.

mapping reduction reduce to decidable problem to prove for decidability for something

Because the language EnglishPalindromes (playing the role we denoted by K in our earlier definitions and theorems) is decidable, we *can* now use a decider for it in order to ensure that strings in EnglishPalindromes are mapped to strings in YearsOfTransitsOfVenus, while strings outside EnglishPalindromes are mapped to strings outside YearsOfTransitsOfVenus. But what do we gain from that? It doesn't give us a new or useful decider for EnglishPalindromes, because it must use a decider that we already have. The language EnglishPalindromes is not undecidable, so it does not help us show that anything else is undecidable.

The illustration we have just given is a special case of a general principle.

Theorem 43
If K is decidable and L is *any* language except $\emptyset$ and Σ^* then $K \leq_m L$.

Proof. Let D be a decider for K.

Let $x^{(\text{yes})}$ be any specific word in L and let $x^{(\text{no})}$ be any specific word in $\overline{L}$.

Mapping reduction from K to L:

Input: a string w

1. Run D on w.
2. If D accepts w then output $x^{(\text{yes})}$
   else output $x^{(\text{no})}$.

This is a theoretical point worth noting, but it does not give us anything useful. In short, there's not much point in a mapping reduction that *decides* K.

Revision

Reading: Sipser, pp. 234–238.

Universal turing machine
↳ stimulate behaviour on any other turing machine

undecidability
↳ cannot be solved by algorithm / computer programs
↳ no program can give correct answer for all possible inputs

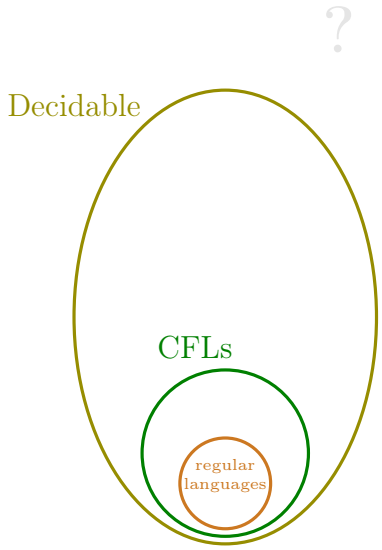
halting problem (undecidable)
↳ given arbitary problem and its input, determine if program halts or run forever
↳ no program / algorithm can solve problems for all possible program
↳ halting problem: return true if halts on input, false otherwise

** undecidability shows the limit of algorithm achievement and hence shows the requirement on certain aspects for human intervention

Lecture 22

Undecidability

Here are the main language classes we have met so far. In this lecture, we explore the region outside the class of decidable languages.



22.1 Undecidable languages exist

We can use countability and uncountability to show that undecidable languages exist.

Theorem 44
Undecidable languages exist.

Proof.
The set of all deciders is countable, because

$$\{\text{CWL-encodings of deciders}\} \subseteq \{\text{CWL}\} \subseteq \Sigma^*$$

and Σ^* is countable. (See Lecture 5.)

It follows that the set of all decidable languages is countable, because each decider defines exactly one decidable language. (In fact, a single decidable language can have many deciders,

but that only helps our argument here. There certainly cannot be *more* decidable languages than there are deciders.)

But we saw in Lecture 5 that the set of *all* languages is uncountable.

Therefore undecidable languages exist. □

In fact, this proof shows that, in a sense, most languages are undecidable! But the proof gives us no clue as to how to actually construct an example of an undecidable language. We give such a construction next. Interestingly, we will make further use of diagonalisation in our first proof of undecidability.

22.2 The Halting Problem

We now give our first example of an undecidable decision problem (and language).

Halting Problem
INPUT: Turing machine P, input x
QUESTION: If P is run with input x, does it eventually halt?

As a language, this is:

$$\text{HaltingProbem} := \{\langle P, x\rangle : \text{when } P \text{ is run with input } x, \text{ it eventually halts.}\}$$

Theorem 45
The Halting Problem is undecidable.

This was proved independently by Alonzo Church (1936), using lambda calculus, and Alan Turing (1936–37), using Turing machines. We will follow Turing's approach.

The main ingredients of the proof are:

- contradiction,
- diagonalisation,
- a version of the Liar Paradox: "This sentence is false."

Before we give the formal proof, we outline the main ideas and show how diagonalisation is used.

Consider what happens when we run Turing machines (encoded as strings) on input strings.

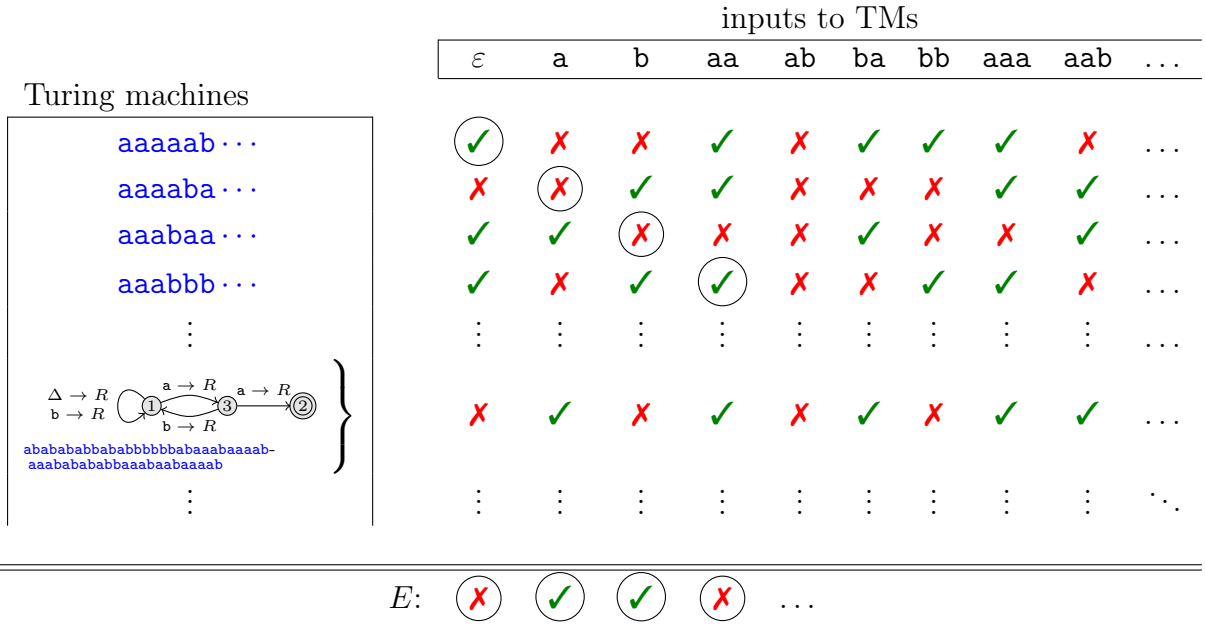
Imagine a large table in which the rows correspond to Turing machines, the columns correspond to all possible input strings, and the entries state whether a given Turing machine halts on a given input string, using ✓ to indicate that it halts and ✗ to indicate that it does not halt.

Part of this table is shown below. You have to imagine infinitely many rows and columns, We have added a double horizontal line below all the rows to keep them from taking over the entire page.

We imagine the encodings of all possible Turing machines listed vertically at the left, with each Turing machine labelling one of the rows. It must be possible to give such a listing: the set of encodings of Turing machines is a subset of the set Σ^* of *all* strings, and we know that the set of all strings is countable. (This is virtually the same as the argument we used above to show that the set of all deciders is countable.) We have not shown the details of the Turing machines, except for one particular row where the Turing machine happens to be one we met on page 198 in Lecture 18.

The columns are labelled by all possible input strings. These strings are listed in the usual order along the top, one per column.

If we look along the row for the diagrammed Turing machine (copied from p. 198) and the column for string ba, we see that the entry in the table is ✓. This means that the Turing machine halts for that input. This agrees with what we worked out on p. 198, that the Turing machine rejects such a string. Remember that Turing machines halt when they either accept or reject an input, and that our marks in this table, ✓ and ✗, are *not* about acceptance versus rejection, but about halting versus looping forever, respectively.



We will use diagonalisation to construct a new Turing machine which is different in its halting-versus-looping behaviour from every Turing machine in our infinite list.

To do this, consider the entries for the i-th Turing machine running on the i-th input string, for each $i \in \mathbb{N}$. These are the circled entries along the main diagonal (top left down towards bottom right) of the table. We want to construct a new Turing machine, E, which disagrees with each of the circled behaviours. For the first input string ε, the first Turing machine halts (✓ in the circled table entry), so we want E to loop forever for this input string (✗). For the second input string a, the second Turing machine loops forever (✗ in the circled table entry), so we want E to halt for this input string (✓). And so on. The desired behaviour of E is summarised along the extra row we have added below the double line

underneath the table. In that extra row, you can see that each circled entry is the opposite of the circled entry above it in the main diagonal of the table.

By construction, E differs from every Turing machine in our infinite list of Turing machines. Specifically, for each $i \in \mathbb{N}$, it differs from the i-th Turing machine in whether it halts or loops forever on the i-th input string. This means that E cannot belong to our list of Turing machines. We will use this deduction to derive a contradiction, because if the Halting Problem is decidable then we can use a decider for it to construct a decider for E.

Those are the main ideas behind the proof that the Halting Problem is undecidable. We now present the proof itself.

Proof.

Assume, by way of contradiction, that there is a Decider, D, for the Halting Problem.

So this decider D can tell, for any P and x, whether or not P eventually halts after being given input x.

So it can tell, for any P, whether or not P eventually halts after being given input P!

Construct another program (Turing machine) E as follows ...

E

Input: P
Use D to determine what happens if P runs on itself.
If D says, "P halts, with input P" : loop forever.
If D says, "P loops forever, with input P" : Halt.

What happens when E is given itself as input?
If E halts, for input E: **then** E loops forever, for input E.
If E loops forever, for input E: **then** E halts, for input E.
This is a ontradiction!
So our original assumption, that the Halting Problem is undecidable, was wrong.
Therefore the Halting Problem is undecidable. □

One of the best explanations of the main ideas of the proof is given in the following YouTube film:

- Udi Aharoni [udiprod],
  Proof That Computers Can't Do Everything (The Halting Problem),
  https://www.youtube.com/watch?v=92WHN-pAFCs

22.3 Other Undecidable Problems

There is an innumerable multitude of other undecidable problems (by the uncountability of the number of languages, compared with the countability of the number of decidable languages, as we saw earlier). So we had better make a start at meeting some of them.

The following problem will prove very useful to us.

DIAGONAL HALTING PROBLEM
INPUT: Turing machine P
QUESTION: Does P eventually halt, for input P?

We can already see that this problem is undecidable, since in our proof of undecidability for the Halting Problem, our assumed decider was only ever used to determine what happens when a Turing machine is run on *itself* as input. In other words, our proof actually served to show that the Diagonal Halting problem is undecidable.

Now let's look at a different type of problem, where we don't ask about anything so exotic as running a Turing machine on itself.

HALT FOR INPUT ZERO
INPUT: Turing machine P
QUESTION: Does P eventually halt, for input 0?

Theorem 46
HALT FOR INPUT ZERO is undecidable.

We'll prove this by mapping reduction from the Diagonal Halting Problem.
Recall that, if there is a mapping reduction f from K to L, then:

Theorem 40:	If L is decidable, then K is decidable.
Corollary 41:	If K is undecidable, then L is undecidable.

So, now that we have a couple of undecidable problems, we can use either of them as a springboard for proving that other problems are undecidable. All we have to do is give a mapping reduction from a known undecidable problem to our new problem, and to prove that it is indeed a mapping reduction of the required type.

For our known undecidable problem, we will use the Diagonal Halting Problem.
We now prove Theorem 46 by this method.

Proof.
Let M be any program, which we regard as an input to the Diagonal Halting Problem.
Define M' as follows:

M'

```
Input: x
Run M on input M.
```

Observe firstly that the construction $M \mapsto M'$ is computable. Given an encoding $\langle M \rangle$ of M, we can write a Turing machine that just simulates the running of M on input $\langle M \rangle$. This can be done by "'hard-coding" a UTM running Turing machine M on input string $\langle M \rangle$.

Furthermore, M halts on input M if and only if M' halts on input 0. (This is because all M' really does is simulate the running of M on input M. So, whatever input M' is given, it eventually halts if and only if M halts on input M.)

So, the function that sends $M \mapsto M'$ is a mapping reduction from DIAGONAL HALTING PROBLEM to HALT FOR INPUT ZERO.

Therefore HALT FOR INPUT ZERO is undecidable. □

There's nothing special about zero, here. We can obtain a whole family of undecidability results by simple tweaks of the problem definition. For example, consider the following problem.

HALT FOR INPUT 42
INPUT: Turing machine P
QUESTION: Does P eventually halt, for input 42?

The proof of its undecidability is virtually identical to the previous one. We can use the same mapping reduction, and just change 0 in the proof to 42.

We can ask other questions about the halting behaviour of a Turing machine.

ALWAYS HALTS
INPUT: Turing machine P
QUESTION: Does P always halt eventually, for any input?

Theorem 47
ALWAYS HALTS is undecidable.

This proof is virtually identical to the previous one.

Proof.

Let M be any program, which we regard as an input to the Diagonal Halting Problem.

Define M' as follows:

M'

```
Input: x
Run M on input M.
```

Observe firstly that the construction $M \mapsto M'$ is computable. Given an encoding $\langle M \rangle$ of M, we can write a Turing machine that just simulates the running of M on input $\langle M \rangle$.

Furthermore, M halts on input M if and only if M' halts for every input. (This is because all M' really does is simulate the running of M on input M. So, whatever input M' is given, it eventually halts if and only if M halts on input M.)

So, the function that sends $M \mapsto M'$ is a mapping reduction from DIAGONAL HALTING PROBLEM to ALWAYS HALTS.

Therefore ALWAYS HALTS is undecidable. □

Here is another problem.

SOMETIMES HALTS
INPUT: Turing machine P
QUESTION: Is there some input for which P eventually halts?

Theorem 48
SOMETIMES HALTS is undecidable.

Proof.
Let M be any program, which we regard as an input to the Diagonal Halting Problem. Define M' as follows:

M'

```
Input: x
Run M on input M.
```

Observe firstly that the construction $M \mapsto M'$ is computable. Given an encoding $\langle M \rangle$ of M, we can write a Turing machine that just simulates the running of M on input $\langle M \rangle$.

Furthermore, M halts on input M if and only if M' halts for some input. (This is because all M' really does is simulate the running of M on input M. So, whatever input M' is given, it eventually halts if and only if M halts on input M.)

So, the function that sends $M \mapsto M'$ is a mapping reduction from DIAGONAL HALTING PROBLEM to SOMETIMES HALTS.

Therefore SOMETIMES HALTS is undecidable. □

The next example doesn't lend itself to a mapping reduction. But we are still able to prove that it is undecidable, using the undecidability of NEVER HALTS and a more general type of reduction in a short proof by contradiction.
NEVER HALTS
INPUT: Turing machine P
QUESTION: Does P always loop forever, for any input?

Observe that NEVER HALTS is the complement of SOMETIMES HALTS.

Theorem 49
NEVER HALTS is undecidable.

Proof.
If D is a decider for NEVER HALTS, then switching Accept and Reject gives a decider for SOMETIMES HALTS.

But we now know that SOMETIMES HALTS is undecidable.
This is a contradiction.
So NEVER HALTS is undecidable too. □

Here are some other undecidable problems. In each case, think about how you would go about proving undecidability. Look for mapping reductions from a problem you already know to be undecidable.

INPUT: Turing machine P and Q
QUESTION: Do P and Q always both halt, or both loop?
i.e., is it the case that:

$$\forall x : P \text{ halts on input } x \iff Q \text{ halts on input } x \qquad \ldots ?$$

INPUT: Turing machine P
QUESTION: If P is run on the input "What's the answer?", does it output "42"?

22.4 Decidable or Undecidable?

Consider each of the following decision problems, and try to determine if it is decidable or undecidable.

INPUT: Turing machine P, input x.
QUESTION: Does P accept x? undecidable

INPUT: Turing machine P, input x, positive integer t
QUESTION: When P is run on x, does it halt in $\leq t$ steps?

INPUT: Turing machine P, positive integer s.
QUESTION: Does P have $\leq s$ states?

INPUT: Turing machine P, positive integer k.
QUESTION: Does P halt for some input of length $\leq k$.

22.5 Undecidable problems about FAs and CFGs

It is not surprising that the next problem is undecidable, since it is asking a more complex question about $\mathsf{Accept}(P)$ than whether it contains $\langle P \rangle$, or 0, or 42, or is empty, or nonempty, or the universal language.

INPUT: a Turing machine P
QUESTION: Is $\mathsf{Accept}(P)$ regular? ...i.e., is P equivalent to a Finite Automaton?

You may have the impression so far that undecidable problems are only about Turing machines, or (more generally) programs for some computer. In fact, there are important undecidable problems about many types of objects that are much simpler than Turing machines. Here are some examples.

INPUT: a CFG
QUESTION: is the language it generates regular? undecidable

INPUT: a CFG
QUESTION: is there any string that it doesn't generate? (over same alphabet) undecidable

INPUT: two CFGs.
QUESTION: Do they define the same language? undecidable

Proofs that the above three languages are undecidable are beyond the scope of this unit. But the fact that these languages are undecidable shows that undecidability reaches across core concepts in computer science and affects some fundamental practical questions.

22.6 Undecidable problems about numbers and strings

In fact, undecidability reaches beyond computer science into some fundamental mathematical questions that do not seem, on the face of it, to be about computational concepts at all.

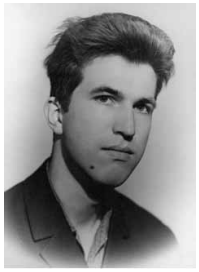
The following problem is about integer roots of multivariate polynomials. It was shown to be undecidable by Yuri Matiyasevich in 1970.[1]

INPUT: a polynomial (in several variables)
QUESTION: Does it have an integer root? undecidable

The **Post Correspondence Problem** is a famous problem about string matching and was shown to be undecidable by Emil Post in 1946[2]

[1] Yuri Matiyasevich, Enumerable sets are diophantine (Russian), *Doklady Academy Nauk, SSSR* **191** (1970) 279–282. English Translation in *Soviet Math Doklady* **11** (1970) 354–358. An accessible exposition of the proof is given in: Martin Davis, Hilbert's tenth problem is unsolvable, *American Mathematical Monthly* **80** (1973) 233–269; reprinted with minor corrections in: Martin Davis, *Computability and Unsolvability*, Dover, New Yourk, 1982, Appendix 2: Hilbert's Tenth Problem is unsolvable, pp. 199-235.

[2] E. L. Post, A variant of a recursively unsolvable problem, *Bulletin of the American Mathematical Society* **52** (4) (1946) 264–269. (see Sipser, Section 5.2). Available at: `https://www.ams.org/bull/1946-52-04/S0002-9904-1946-08555-9/S0002-9904-1946-08555-9.pdf`.



https://mathshistory.st-andrews.ac.uk/Biographies/Matiyasevich/

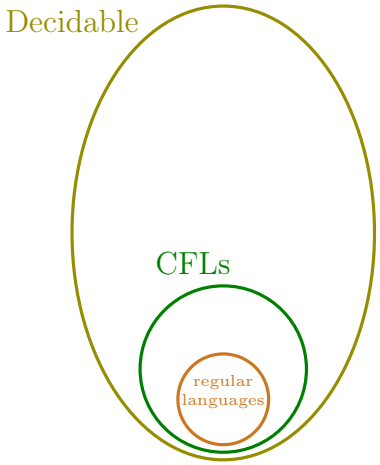
Yuri Matiyasevich (b. 1947)

https://mathshistory.st-andrews.ac.uk/Biographies/Post/

Emil Post (1897–1954)

22.7 Language classes

Here are our language classes again, with no question mark outside the class of Decidable languages, since we now know that that region is nonempty (and, in fact, vast).



Revision

Reading: Sipser, pp. 201–209, 215–220, 234–236.

Preparation: Sipser, pp. 170, 209–210.

HALTING PROBLEM

↳ decision problem
↳ determine if it halts or run forever
↳ undecidable

undecidable proving: Alan Turing proof

① Assumption: there exists a program that decide the halting problem
② 2 program: HaltingDecider() & Program1()

- HaltingDecider() halts when Program1() runs forever
- HaltingDecider() runs forever when Program1() halts

③ new program introduced: DeciderDecision() } scenario 2
+ Program1() runs on itself } scenario 1

scenario 1:
- Program1() halts with Program1() as input
  ↳ HaltingDecider() runs forever when Program1() halts
- Program1() runs forever with Program1() as input
  ↳ HaltingDecider() halts when Program1() runs forever

scenario 2:
- DeciderDecision() halts with DeciderDecision() as input
- DeciderDecision() runs forever with DeciderDecision() as input

CONTRADICTION!!!

scenario 1:
- If Program1() halts then HaltingDecider() runs forever
- If Program1() runs forever then HaltingDecider() halts

Nothing is decided, it contradict itself. Program1() run on itself as input, if halted then HaltingDecider() runs forever, if runs forever then HaltingDecider() halts. HaltingDecider() is always wrong.

scenario 2:
DeciderDecision() was assumed to correctly solve Halting problem. DeciderDecision() with itself as input. If DeciderDecision() halts then DeciderDecision() runs forever. If DeciderDecision() runs forever then DeciderDecision() halts. It makes no sence so DeciderDecision() can't correctly solve itself to halt or run forever so DeciderDecision() is undecidable so there exist no general algorithm that can correctly predict another program that might be itself to halt or run forever.

HALT FOR INPUT ZERO (EMPTY STRING)

↳ program halts for no input / empty string
↳ undecidable

Assumption: There exist no program that can decide whether an arbitrary program will halt when run on no input

① show if can halt for zero (mapping reduction by example)

- program HaltZero() decides if program1() can halt for zero.
- program GeneralHalt() decide on general halting problem.

HaltZero() halts when program1() halts for zero.
GeneralHalt() check if HaltZero() really halts.
↳ GeneralHalt() halts when HaltZero() halts.
↳ GeneralHalt() runs forever when HaltZero() runs forever.

* GeneralHalt() halts when if and only if HaltZero() halts but HaltZero() only halts if and only if program1() halts for zero.

GenaralHalt() is an extension of the general halting problem. General halting problem is undecidable so the haltzero() is also undecidable.

Hence, the problem on halt for input zero is also undecidable.

② mapping reduction (ACTUAL)
HaltZero() halts when program1() halts for zero as input.

- Given program P and input x, there exists program P' that takes input $x=0$ (no input).
- HaltZero() check if P' halts when $x=0$. If P' halts then P also halts for input $x=0$.

- If HAH when input zero is decidable then halting problem is also decidable but since halting problem is undecidable so using mapping reduction, HAH when input zero is also undecidable.

Lecture 23

there exist an algorithm that can recognise whether a solution exist
BUT this algorithm may not always halt/ provide definite answer

Recursively enumerable languages

Having first moved beyond decidability in Lecture 22, we continue that journey by exploring a class of languages that are "partially decidable" in a precise sense. This is the class of *recursively enumerable* languages. This same class captures the power of Turing machines to *generate* languages, which is greater than their power to *decide* languages. But even this class has its limits: we will meet languages that are not even recursively enumerable.

23.1 Relaxing decidability

Recall that a language L is decidable if and only if there exists a Turing machine T such that

$$\begin{aligned} \mathsf{Accept}(T) &= L \\ \mathsf{Reject}(T) &= \overline{L} \\ \mathsf{Loop}(T) &= \emptyset. \end{aligned}$$

We now explore what happens when we relax this definition to give a kind of one-sided version of decidability.

A language L is **recursively enumerable** if there exists a Turing machine T such that

$$\mathsf{Accept}(T) = L$$

re = recursively enumerable

Strings outside L may be *rejected*, or may make T *loop forever*.

It is common to write **r.e.** for *recursively enumerable*.

Some synonyms for recursively enumerable include **computably enumerable**, **partially decidable**, **Turing recognisable** (which is used in Sipser) and **type 0** (which is one of the language class in the *Chomsky hierarchy*.

The term "computable" is also sometimes used as a synonym for r.e., but that can be confusing as it is also used for "decidable".

23.2 Decidable versus r.e.

**

It is clear from the definitions that every decidable language is recursively enumerable.

It is natural to ask, is every recursively enumerable language decidable?

Consider:

$$\text{HALT} = \{T : T \text{ halts, if input is } T\}.$$

This is the language corresponding to the Halting Problem. We know it's not decidable. Is it recursively enumerable?

Let M be a Turing machine which takes, as input, a Turing machine T and

- simulates what happens when T is run with *itself* as its input.
- If T stops (in any state, either an Accept state or a Reject state or by crashing), M accepts.

Here, M could be obtained by modifying a UTM.

We find that:

$$\begin{aligned} \mathsf{Accept}(M) &= \text{HALT}, \\ \mathsf{Reject}(M) &= \emptyset, \\ \mathsf{Loop}(M) &= \overline{\text{HALT}}. \end{aligned}$$

potentially recognised process when halt on specific input BUT MAY NOT TERMINATE/ PROVIDE ANSWERS IN CERTAIN CASES

HALT is recursively enumerable BUT NOT DECIDABLE

So HALT is recursively enumerable.

So some recursively enumerable languages are not decidable.

Consider the list of undecidable languages given in Lecture 22. Which ones are recursively enumerable?

We introduced recursive enumerability as a "one-sided" version of decidability. With this thought in mind, we might expect that ordinary "two-sided" decidabilty might be obtained if both a language and its complement are r.e. This is indeed the case, and we formalise it in the next theorem.

Theorem 50

A language is decidable if and only if both it and its complement are r.e.

Proof.

If always halt / terminate = Decidable
↳ else, sometimes halt / terminate, sometimes not = recursively enumerable

($\Longrightarrow$)

Let L be any decidable language.

We have seen that every decidable language is r.e. So L is r.e.

Now, the complement of a decidable language is also decidable. (See our comments on closure properties of the class of decidable languages, on page 228 in Lecture 20.)

So $\overline{L}$ is also decidable, and therefore also r.e.

So L and $\overline{L}$ are both r.e.

($\Longleftarrow$)

Let L be any language such that both L and $\overline{L}$ are r.e.

Since they are each r.e., there exist Turing machines M_1 and M_2 such that

$$\begin{aligned} \mathsf{Accept}(M_1) &= L \\ \mathsf{Accept}(M_2) &= \overline{L}. \end{aligned}$$

HALTING PROBLEM

Program stimulates execution of program on given input → If halt, program accepts

→ If don't halt (run forever), program may not reject nor provide definite answer

may not always terminate / halt = NOT decidable

Why R.E.? can potentially recognise halting → halt & accepts when program halts BUT may not always halt when program runs forever

Note, each of these TMs might *loop forever* for inputs they don't accept.

Construct a new Turing machine M' that simulates *both* M_1 and M_2:

Input: x

Repeatedly:

Do one step of M_1. If it **accepts**, then Accept.

Do one step of M_2. If it **accepts**, then Reject.

Before continuing with the proof, we comment on the way this algorithm works. Since we want to simulate both M_1 and M_2, it would be tempting to just run M_1 first, then (once M_1 has finished) run M_2. The trouble with this is that M_1 might not finish, so we might never have the opportunity to run M_2 to see if it accepts (and we don't actually *know* if M_1 won't *ever* finish). The same issue arises, the other way round, if we try running M_2 all the way followed by M_1. We don't know in advance which of M_1 or M_2 might accept, and whichever one doesn't accept might loop forever. So we don't know in advance which of M_1 or M_2 to run first. We'd like to run M_1 and M_2 in parallel, perhaps on separate Turing machines that run simultaneously. But, in order to get a decider, we have to show how to do the whole thing on a *single* Turing machine, without parallelism. In the method given above, we *interleave* the simulations of M_1 and M_2 on a single Turing machine, by taking turns to simulate successive steps of each. This is called a **time-sharing simulation**. It requires quite a lot of record-keeping on the way, to keep track of everything that's happening. At any time, a record needs to be kept, for each of the two Turing machines, of the machine's current state, tape cell, and tape contents. Managing all this requires care but is certainly doable.

M' is a decider, because

- every string belongs to either L or $\overline{L}$,
- therefore is accepted by either M_1 or M_2,
- therefore will eventually be either accepted or rejected by M'.

Furthermore, M' accepts x if and only if M_1 accepts x.
So M' is a decider for L.
So L is decidable. □

23.3 A non-r.e. language

Several times now, we have defined new classes of languages that contain all classes we have previously met as well as some languages that did not belong to our previous classes. With each of these new classes, we have asked, *is that the lot?* We do the same again now.

Is *every* language recursively enumerable?

Consider:

$$\overline{\text{HALT}} = \{T : T \text{ loops forever, if input is } T\}$$

Theorem 51
$\overline{\text{HALT}}$ is not r.e.

Proof.
Assume $\overline{\text{HALT}}$ is r.e.
We already know that HALT is r.e.
So, both HALT and its complement are r.e.
Therefore, by Theorem 50, HALT is decidable.
This is a contradiction!
Therefore $\overline{\text{HALT}}$ is not r.e. □

23.4 Enumerators

In our studies of Turing machines so far, we have used Turing machines as tools to help us decide, in some cases at least, whether a given string belongs to a particular language. But Turing machines can also produce output, as we saw in Lecture 18. So we could use Turing machines to *output* the strings in a language. For a Turing machine to be able to output *all* the strings in a language, we will usually need to let it run forever, since most languages that interest us are infinite.

An **enumerator** is a Turing machine which outputs a sequence of strings. The Turing machine has a special Output State which works as follows. Whenever the TM enters its output state, the string on the tape up to, but not including, the first blank is taken to be the next output string. At the next step, the TM is allowed to go to another state. The TM never accepts or rejects; it just keeps outputting strings, one after another.

The sequence of strings output by the enumerator can be finite or infinite. If it's infinite, then the enumerator will never halt. If the sequence is finite, then the enumerator may stop once it has finished outputting. But the state it enters doesn't matter.

A language L is **enumerated** by an enumerator M if

$$L = \{\text{all strings in the sequence outputted by } M\}$$

Members of L may be outputted in any order by M, and repetition is allowed.

Theorem 52
A language is recursively enumerable if and only if it is enumerated by some enumerator.

Proof.
($\Longleftarrow$)
Let L be a language, and let M be an enumerator for it.
Construct a Turing machine M' as follows:

Input: a string x

Simulate M, and for each string y it generates:

Test if $x = y$. If so, accept; otherwise, continue.

A string x is accepted by M' if and only if it is in L.
So $\mathsf{Accept}(M') = L$. So L is r.e.

($\Longrightarrow$)
Let L be r.e. Then there is a TM M such that $\mathsf{Accept}(M) = L$. Take all strings, in order:

$$\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \ldots$$

Simulate the execution of M on each of these strings, simultaneously.
As soon as any of them stops and accepts its string, we pause our simulation, output that string, and then resume the simulation.

Before we proceed, we must be *careful*. We have infinitely many executions to simulate, but we only have finite time! How do we schedule all these simulations?
Denote the strings by $x_1, x_2, \ldots, x_i, \ldots$ (So $x_1 = \varepsilon$, $x_2 = \mathtt{a}$, etc.)
Here is an algorithm to do all the required simulations. This is another example of a *time-sharing simulation*.

For each $k = 1, 2, \ldots$
  For each $i = 1, \ldots, k$:
    Simulate the next step of the execution of M on x_i
      (provided that execution hasn't already stopped).
    If this makes M accept, then
      output x_i and skip i in all further iterations;
    else if this makes M reject, then
      output nothing, and skip i in all further iterations.

This algorithm can be implemented by a Turing machine.
Any string accepted by M will eventually be output.
So this is an enumerator for L. □

This theorem explains the term "recursively enumerable" (and "computably enumerable").
It also explains why r.e. languages are sometimes called *computable*, since there is a computer that can *compute* all its members (i.e., can generate them all).

23.5 Verifiers

Another view of r.e. languages is given by the following theorem.

Theorem 53
A language L is r.e. if and only if there is a decidable two-argument predicate P such that

$$x \in L \iff \exists y : P(x, y).$$

This P is a *verifier*: if you are given y then you can use P to *verify* that x is in L (if it is).

But it may be hard to *find* such a y.

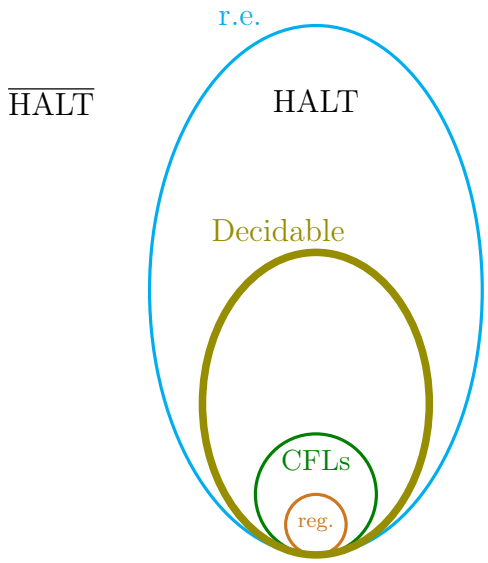
We established the relationship between decidable languages and mapping reductions in Theorem 40. We now do the same for r.e. languages.

Theorem 54
If $K \leq_m L$ and L is r.e. then K is r.e.

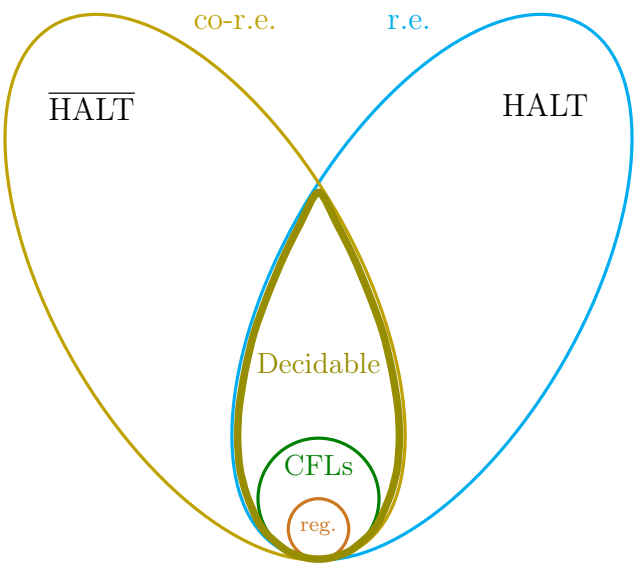
→ mapping reduction

23.6 Language classes

Here is the latest edition of our Venn diagram for language classes, showing the specific languages HALT and $\overline{\text{HALT}}$ within (or outside) the appropriate classes.



A language is **co-r.e.** if its complement is r.e. We have seen that — unlike the class of decidable languages — the class of r.e. languages is *not* closed under complement (see Theorem 50). So, with both the class of r.e. languages and the class of co-r.e. languages, our landscape of language classes becomes as follows.



Revision

Reading: Sipser, pp. 170, 209–211.

Preparation: Sipser, pp. 275–286.

Lecture 24

Polynomial time, and the class P

After a journey high into the stratosphere of language classes — encountering languages that are "theoretically" decidable and even some that are not — it is time to come down to earth and consider what it means for a language to be "practically" or "efficiently" decidable. To this end, we focus on languages that are decidable using running time that does not grow too fast. We define a language class to try to model this, consider its properties, and meet many of its members.

24.1 Decidability and computational resources

Decidable languages are "solvable", in principle, by a computer. It doesn't matter (much) which definition of "*computer*" you use. The set of *decidable languages* is the same.

BUT what about the resources required? The concept of decidability pays no attention at all to how much time (as modelled by the number of steps the Turing machine takes) or space (i.e., memory, as modelled by the number of tape cells used) is used. The TM can take arbitrarily long and use an arbitrarily large amount of space, as long as it halts eventually.

The usage of computational resources is of great practical importance. So we need to look at what it means for a language (or problem) to be solved efficiently, in practice.

The most important computational resource is time. So we will focus on that.

24.2 Time complexity

For any Turing machine M, define:

$$\begin{aligned} t_M(x) &:= \text{time taken by } M \text{ for input } x \\ &:= \#\text{ steps } M \text{ takes until it halts.} \end{aligned}$$

The **time complexity** of M is the function defined for positive integer arguments by

$$\begin{aligned} t_M(n) &:= \text{maximum time taken by } M \text{ for any input of length } n \\ &= \max\{\, t_M(x) : |x| = n \,\}. \end{aligned}$$

Time complexity is a worst case measure, in that it gives the *maximum* amount of time that M needs over all inputs of a given length.

For it to be defined for a given n, the TM M must halt on all inputs of length n.
For it to be defined for all n, the TM M must halt for all inputs, i.e., it must be a decider.

Examples

- Turing machine to decide whether a string ends in `b`
  - moves to the right until reaches first blank symbol,
    then does one step to the left and
    accepts if the symbol there is `b`, otherwise rejects
  - time complexity? $n + 1$

- Turing machine to decide whether a string starts and ends with the same letter
  - time complexity? $2n + 1$

- Turing machine to decide whether a string is a palindrome
  - time complexity? $\approx \frac{1}{2}(n + 1)(n + 2)$

- Turing machine to decide whether a string is empty
  - time complexity? 1

Exercise:
Consider any regular language. What can you say about its time complexity?
See Theorem 32 in Lecture 18 on Turing Machines:

$$\text{Finite Automaton} \longrightarrow \text{Turing machine}$$

Time complexity depends on the type of Turing machine (or computer) used.
Turing machine details that affect time taken include:

- the number of symbols used,
- whether tape is infinite in both directions (or just one),
- the number of tapes,
- the dimensionality of "tape" (1-D, 2-D, ...?),
- whether the allowed movement directions include sitting still,
- ...

efficiently solvable ⇒ run in polynomial time

In FIT1045 (and FIT1008 too if you've done it), you may have met time complexity of algorithms and programs.

Those time complexities still assumed some theoretical computer (maybe implicitly), and can be sensitive to the assumptions made.

We will use time complexity to formalise the notion of "efficiently solvable".

But setting a specific threshold (e.g., $3n^5$) is too arbitrary. The meaning of "efficiently solvable" would then either be tied to a specific type of Turing machine (computer) or it would change with time (as technology improves).

We would like to capture "efficiently solvable" in a way that is more robust to the computer (or model of computation) used and less vulnerable to technological change. We hope to identify problems where "efficient solvability" is *a characteristic of the problem itself*, rather than of the tools and technologies we happen to have at the time.

In Table 24.1, we consider some simple functions that can arise as time complexities of some algorithms. The first column gives a sequence of possible input sizes, initially increasing by an *amount* of 10 from one row to the next, then showing increases by *factors* of 10 further down. The time complexity functions are given as the labels of the other column: n, n^2, n^3, n^4, 2^n, 10^n. To see how a given time complexity function grows as n grows, you can look down its column.

The first four time complexities are *polynomial* time complexities, because they are fixed powers of n. The last two are *exponential* time complexities, because they each have a fixed base (2 or 10) with n in the exponent.

Looking down the various columns, you can see a striking difference between the behaviours of polynomial and exponential time complexities. See how the times grow down the first few (polynomial) columns, versus how they grow down the last two (exponential) columns. The growth down the first few entries of the exponential columns is so rapid that the numbers get a few more digits whenever n increases by 10. Nothing like that kind of growth occurs for the polynomial time complexities. And, when n is *multiplied* by 10, the exponents in the exponential times get one digit longer each time. Again, this is way beyond anything we see for the polynomial time complexities.

Table 24.2 summarises some general observations comparing polynomial time complexities of the form n^c with exponential time complexities of the form c^n, where c is constant in each case. There is always a stark contrast between the implications of change for polynomial and exponential times. We would clearly much rather have algorithms with polynomial time complexities than exponential time complexities.

24.3 Polynomial time

A Turing machine M has **polynomial time complexity** if its time complexity is $O(n^k)$, for some fixed k.

$$t_M(n) = O(n^k).$$

This means that there is a constant c such that, for all *sufficiently large* n,

$$t_M(n) \leq c \cdot n^k.$$

input size n	time n	n^2	n^3	n^4	$\cdots$	2^n	10^n
10	10	100	1000	10000		1024	10^{10}
20	20	400	8000	160000		1048576	10^{20}
30	30	900	27000	810000		1073741824	10^{30}
40	40	1600	64000	2560000		1099511627776	10^{40}
$\vdots$					$\cdots$		
100	10^2	10^4	10^6	10^8		$\approx 10^{30}$	10^{100}
$\vdots$							
1000	10^3	10^6	10^9	10^{12}		$\approx 10^{300}$	10^{1000}
$\vdots$							
10000	10^4	10^8	10^{12}	10^{16}		$\approx 10^{3000}$	10^{10000}

Table 24.1: Comparison of time complexities

If you ...	n^c time (polynomial)	exponential time
increase input size by a fixed *amount* ... (i.e., $n \longrightarrow n + k$)	... then time increases by an amount $O(n^{c-1})$	... then time increases by a fixed *factor*
increase input size by a fixed *factor* k ... (i.e., $n \longrightarrow kn$)	... then time increases by fixed *factor* k^c	... then time is raised to *power* k
double your computer's speed ...	... then you increase feasible input size by some fixed *factor*.	... then you increase feasible input size by some fixed *amount*.
need to handle inputs which are twice as large as those you can solve now ...	... then you must wait for $2c$ years before computers are fast enough. *	... then your wait is proportional to your current input size. *

* assumes Moore's Law: processor speed doubles every two years.

Table 24.2: Comparison of polynomial time and exponential time under various scenarios.

Symbolically:

$$\exists c\, \exists N\ \forall n\ :\ n > N \implies t_M(n) \leq c \cdot n^k.$$

The power, k, is fixed. It does not depend on the input. But different polynomial time Turing machines often have different k.

Note that we do not require that the time complexity function $t_M(n)$ be *equal* to a polynomial in n (so the time complexities in the table above were idealised; real time complexities are never so simple). Time complexity functions can be very complicated because of all the intricate detail in the way Turing machines work. We will seldom be interested in finding an *exact* formula for a time complexity. We are only interested in how the time complexity grows as n becomes very large. For it to qualify as *polynomial* time complexity, it has to have an *upper bound* that grows like a polynomial for sufficiently large n.

Although we talk about *polynomial* time, we are really only interested in the leading terms of polynomials that bound time complexity. Suppose our time complexity $t_M(n)$ is bounded above by a polynomial function $a_0n^k + a_1n^{k-1} + \cdots$. As n grows, the leading term a_0n^k will eventually dominate. We will always be able to find a constant b such that, for large enough n, our time complexity is bounded above by bn^k. This means that $t_M(n) = O(n^k)$.

24.4 The class P

A language is **polynomial time decidable** if it can be decided by a polynomial time Turing machine.

The class of all languages decidable in polynomial time is called P (which stands for Polynomial time).

This is the simplest (and, historically, the first) formal notion of "efficiently solvable".

We can look at the simple time complexities we worked out before and verify that the languages considered are all in P.

{strings that end in b}	time complexity	=	$n+1$	=	$O(n)$	✓
{strings that start & end with the same letter}	time complexity	=	$2n+1$	=	$O(n)$	✓
{palindromes}	time complexity	=	$\frac{1}{2} \cdot (n+1)(n+2)$	=	$O(n^2)$	✓
$\{\varepsilon\}$	time complexity	=	$O(1)$	=	$O(n^0)$	✓
any regular language	time complexity	=	...			✓
any context-free language	time complexity	=	...			✓

24.5 The class P: properties

P has been defined using a particular type of Turing machine M (two symbols, single one-way-infinite tape, moves one step left or right, ...).

But what if we used a different type of machine? Or some other model of computation? (E.g., your laptop, a smartphone, CSIRAC, TaihuLight, LUMI, Fugaku, Frontier, ...)

Would we get a different class P ?

Theorem 55
Suppose that

- computer M_1 has time complexity $t(n)$.
- computer M_2 can simulate machine M_1. (E.g., M_2 is a UTM.)
- any computation that takes time t on M_1 takes time $\leq ct^k$ on M_2. (polynomial slowdown)

If M_1 is polynomial time, then M_2 takes polynomial time to simulate M_1 on input strings for M_1.

Proof.
If M_1 takes polynomial time, then $t(n) = O(n^K)$ for some fixed K.
In other words, there exist c_1 and K such that $t(n) \leq c_1 n^K$ for sufficiently large n.
The time taken by M_2 to simulate M_1 on an input of size n is

$$\begin{aligned}
&\leq c \cdot t(n)^k \\
&\leq c\,(c_1 n^K)^k \quad \text{for sufficiently large } n \\
&\leq c\,c_1^k\, n^{Kk} \\
&\leq c'\, n^{k'}, \quad \text{where } c' = cc_1^k \text{ and } k' = Kk \quad \text{(note, both constants)} \\
&= O(n^{k'}).
\end{aligned}$$

□

It follows that, for such M_1 and M_2: if a language L can be decided in polynomial time using M_1, then it can be decided in polynomial time using M_2.

In fact, virtually any two computers can play the roles of M_1 and M_2 here. This is because any "reasonable" computer can simulate any other computer with at most polynomial slowdown.

So the class P is independent of the particular model of computation used to define it.

This is reminiscent of the history of decidability: different paths can be taken to formulate the definition, using different models of computation, but it turns out that they all lead to the same class of decidable languages.

The class P was formulated in work done independently by Alan Cobham (1965), Jack Edmonds (1965) and Michael Rabin (1966). There is thus an intriguing historical analogy with the first studies of computability. In each case, it seemed to be an idea whose time had come.

Here are some examples of members of P.

- the set of pairs of strings in lexicographic order
- the set of strings of matching parentheses
- the set of pairs of numbers that are coprime (a.k.a. relatively prime) [Euclidean algorithm]
- the set of square numbers
- the set of prime numbers (Agrawal, Kayal, Saxena, *Annals of Mathematics*, 2004)
- the set of invertible matrices [MAT1841/MAT2003]
- the set of trees
- the set of balanced binary trees
- the set of connected graphs:

$$\{(G, s, t, k) : G \text{ has an } s\text{–}t \text{ path of length} \leq k\}$$

- the set of regular graphs (i.e., all vertices have the same degree)
- the set of 2-colourable graphs (graphs whose vertices can each be coloured Black or White, so that adjacent vertices receive different colours)
- the set of Eulerian graphs [MAT1830/FIT1045]
- the set of planar graphs (i.e., graphs which can be drawn in the plane so that no two edges cross, except possibly at their endpoints) [advanced]

The next example we give is important in its own right and also because it helps introduce some other languages that are central in the study of computational complexity.

2-SAT := { satisfiable Boolean expressions in Conjunctive Normal Form (CNF), with two literals per clause }.

The "2-" in the name indicates that every clause has exactly two literals.
Here is an example of a member of 2-SAT.

$$(\neg x \vee \neg y) \wedge (x \vee \neg z) \wedge (y \vee z) \wedge (y \vee \neg y)$$

A **truth assignment** is an assignment of a truth value to each of the variables, i.e., a function $f : \{\text{ variables }\} \to \{\,\mathsf{True}, \mathsf{False}\,\}$.

For example,

$$f(x) = \mathsf{False}, \quad f(y) = \mathsf{False}, \quad f(z) = \mathsf{True}$$

is a truth assignment for the above Boolean expression. It gives the expression the value

$$(\neg F \vee \neg F) \wedge (F \vee \neg T) \wedge (F \vee T) \wedge (F \vee \neg F) \;=\; \ldots \;=\; \mathsf{False}.$$

Do all truth assignments make this expression False?

An expression is **satisfiable** if it has a truth assignment which makes the expression True.

For example, the above expression is satisfiable, since the truth assignment

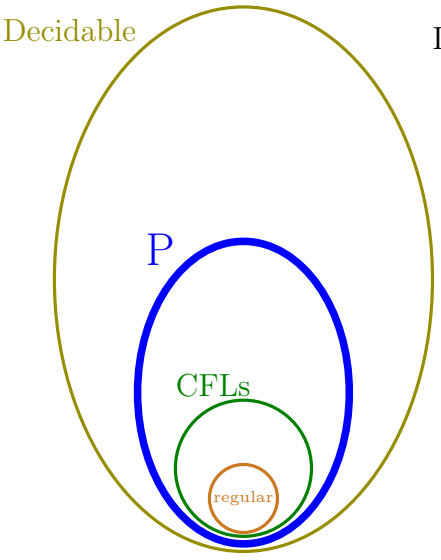
$$\begin{aligned} g(x) &= \mathsf{True} \\ g(y) &= \mathsf{False} \\ g(z) &= \mathsf{True} \end{aligned}$$

makes the expression True.

Challenge: show that 2-SAT is in P.

24.6 P and other language classes

We now update our diagram of language classes.



Decidable, not in P:

Time-bounded Halting Problem, Generalised Go,
Regexp Equivalence, Generalised Chess,
...

In P, not in CFL:

2-SAT, 2-COLOURABILITY,
Invertible matrices, CONNECTED GRAPHS,
PRIMES, SHORTEST PATH,
$\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 1\}$, EULERIAN CIRCUIT,
... ...

In CFL, not Regular:

$\{\mathtt{a}^n\mathtt{b}^n : n \geq 1\}$, PALINDROMES, DYCK, ...

Regular:

EVEN-EVEN, Multiples of 3, CWL, ...

Revision

Reading:

- Sipser, sections 7.1–7.2.

The following classic is also worth reading.

- M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., San Francisco, 1979, Chapter 1 and especially the first two sections of Chapter 2: §2.1, §2.2.

Question: show that 2-SAT is in P

Determine if there exist assignment of truth values to the variable that makes the entire expression true

algorithm to solve 2-SAT: ① construct implication graph
② find strongly connected component
③ check for contradiction
④ satisfiability

* no contradiction found = satisfying contradiction provided

It is in P because it exists an efficient algorithm that's solved in polynomial time

① constructing implication graph based on 2-SAT expression
↳ linear time $o(n)$
↳ n = no. of clauses in 2-SAT expressions

② found using algorithm (Kosaraju / Tarjan)
↳ linear time $o(n)$

③ check for contradictions
↳ linear time $o(n)$

④ satisfiability assignment
↳ linear time $o(n)$
↳ X contradiction = algorithm construct satisfying assignment

overall time complexity = linear time $o(n)$

linear time complexity = bound by polynomial function

algorithm 2-SAT solve problem in polynomial time = falls within class P

Lecture 25

Nondeterministic Polynomial time, and the class NP

There is a huge class of problems of great practical importance for which a solution can be efficiently *checked*, once it is found, but which seem to be hard to *solve*. In this lecture we introduce the class NP in order to study languages with this property. We discuss its relationship with the class P, introduced in Lecture 24, and the famous P-versus-NP problem, which has remained open for over half-a-century.

25.1 Deciding and Verifying

A polynomial-time decider for a language L determines, in polynomial time, whether or not an input string x belongs to the language. It does so *from scratch*, i.e., without the aid of any additional information.

We now look at *polynomial-time verifiers*, which can verify that an input string belongs to a language with the aid of some extra information which we call a *certificate*.

To set the scene, suppose you are interested in the set of people who can kick a football.

How do you verify that a person can kick a football? You could give them a ball and get them to try to kick it. This procedure is like a decider. It enables you to *decide* whether or not they can kick a football.

Now suppose you are interested in the set of university graduates.

How do you verify that a person is a university graduate? You can't do it just by meeting them, testing their abilities, and so on. You could make enquiries of universities, but *without any further information*, you may have a lot of work to do: there are thousands of universities in the world. So, there is really no efficient decider for this set.

But, if you have a person's degree certificate, you can, in principle, verify that they are a university graduate (assuming you have the ability to check that the certificate is theirs and that it is an authentic document). This gives an efficient verifier for the set of university graduates.

All we require, to verify that someone is a university graduate, is *one* valid degree certificate for the person. It does not matter that other certificates might lead to rejection: if, for example, you use their swimming proficiency certificate, or a *different* person's degree

certificate, then the verification process fails, but that doesn't mean they are not a graduate. In such cases, the fault lies not with the verification procedure but with the certificate that was supplied to it. easy to verify the truth but not the fault (?)

While this verification procedure enables you to verify (with the aid of a certificate) that someone is a graduate, the situation is one-sided: it is still hard to verify that someone is not a graduate.

We now define verifiers for languages.

A **verifier** for a language L is a TM that takes, as input, two strings x and y, and

- always halts;
- if x is in L, there exists y such that the TM accepts;
- if x is not in L, every y makes the machine reject.

The string y is called a **certificate**.

For a given input x, a verifier can have many different possible computations. Each certificate gives rise to a potentially different computation for that input. Some of these computations may lead to acceptance, others may lead to rejection. All we require, for strings in L, is that *some* certificate leads to acceptance by the TM. For strings not in L, *no* certificate can possibly lead to acceptance.

A **polynomial-time verifier** is a verifier with time complexity polynomial in n, where $n = |x|$. So a polynomial-time verifier has time complexity $O(n^k)$ for some fixed k.

25.2 NP

NP is the set of languages for which there is a polynomial-time *verifier*.

NP stands for Non-deterministic Polynomial time (for reasons to be given later, in §25.8).

NP is intended to contain languages for which membership can be efficiently verified, with the aid of an appropriate certificate.

25.3 Proving membership of NP

To show a language is in NP, you need to:

- specify the *certificate*,
- give a polynomial-time verifier (as an *algorithm*),
- *prove that it is a verifier* for the language,
- prove that it is *polynomial time*.

We illustrate this by proving that the set of 3-colourable graphs is in NP. (We have already met the set of 2-colourable graphs, which is in P (see page 263 in Lecture 24). But the set of 3-colourable graphs is not known to be in P.)

Theorem 56
{ 3-colourable graphs } is in NP.

Proof.

We must first specify the certificate. For a given graph G, the certificate will be a function $f : V(G) \to \{\text{Red}, \text{White}, \text{Black}\}$ which assigns, to each vertex of G, one of the three available colours.

Now we give the verifier, as an algorithm.

For each edge uv of G
{

Look up $f(u)$ and $f(v)$ in the certificate.

// ... these are the colours given to the endpoints u, v of this edge

Check that $f(u) \neq f(v)$.

If so, continue. If not, Reject and halt.

// ... endpoints must get different colours

}
If loop completes with no edge rejected, then Accept and halt.

Now we must prove that this is a verifier the language and that it runs in polynomial time.

Claim 1:

This is a verifier for { 3-colourable graphs }.

Proof of Claim 1:

G is in { 3-colourable graphs}
if and only if there exists a function $f : V(G) \to \{\text{Red}, \text{White}, \text{Black}\}$ such that, for each edge uv, we have $f(u) \neq f(v)$
if and only if there exists a certificate such that our verifier accepts G.

End of proof of Claim 1.

Claim 2:

The verifier takes polynomial time, in the size of input.

Proof of Claim 2:

From studying the algorithm, we see that the number of main loop iterations equals the number of edges, which we denote by m.

For each edge, we must look up each endpoint in the certificate. Suppose the certificate is given as a list of colours, one for each vertex, with the vertex giving the position in the list.

Then looking up the colour of each endpoint takes $O(n)$ time, where $n := \#$ vertices.
Once we have these two colours, checking whether $f(u) \neq f(v)$ takes constant time.
So, the total time taken is $\leq m \cdot n \cdot \text{constant} = O(mn)$.
So it takes polynomial time, in the size of G.

End of proof of Claim 2.

So we have proved that $\{$3-colourable graphs$\}$ is in NP. □

Remarks:

- Some of these time estimates are loose upper bounds.
- Better estimates are often possible. (E.g., how long does it take to look something up in an array of size n?)
- But if our objective is to show that something is in NP, then all we need to show is that the time complexity of verification is bounded above by a polynomial (i.e., that it is $O(n^k)$, for some fixed k).

25.4 Some languages in NP

We now give a list of examples of languages in NP. For each of them, you should consider:

- What is the certificate?
- How do you verify it?

Examples:

certificate: valid 2-colouring of graph
verification process: check each node is assigned one of 2 colours & adjacent nodes have different colours
↳ done in polynomial time

- the set of 2-colourable graphs

same though process

- the set of 3-colourable graphs

certificate: valid k-colouring of graph G
verification process: verify each node is assigned on one of the k-colours & adjacent nodes have different colours

- $\{(G, k) : G \text{ is a } k\text{-colourable graph}\}$

** - the set of composite numbers

$$\{ x \in \mathbb{N} \ : \ \exists y, z \in \mathbb{N} \text{ such that } 1 < y < x,\ 1 < z < x, \text{ and } x = y \cdot z \}$$

done in polynomial time

- SATISFIABILITY:
  the set of satisfiable Boolean expressions in Conjunctive Normal Form
- 2-SAT

certification: valid truth assignment to variables of Boolean expression that makes evaluation = TRUE

  - This is the restriction of SATISFIABILITY to expressions with exactly two literals in each clause.

certificate: valid truth assignment to variables of 2-SAT expression

  - See p. 263 near the end of Lecture 24.

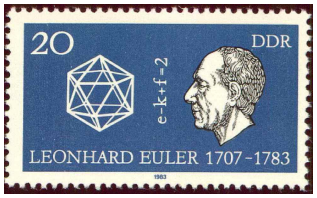
verification process:
check whether the certificate makes the expression TRUE by substituting truth assignment from certification into the CNF expression and evaluate if it = TRUE
↳ valid certification
↳ done in polynomial time

verification process:
substitute truth assignment to 2-SAT expression & check if at least one clause in each literal pairs evaluates to TRUE
↳ done in polynomial time

verification process: check whether certificate makes the expression true by substituting truth variable to 3-SAT expression then check if one literal in each clause evaluates to TRUE

- 3-SAT
  certificate: valid truth assignment to variables of 3-SAT expression that satisfies the expression
  - This is the restriction of SATISFIABILITY to expressions with exactly three literals in each clause.
- the set of Eulerian graphs
  certificate: a valid eulerian tour or circuit in the graph
  - An **Euler tour** in a graph G is a closed walk along edges that uses each edge exactly once.
  - A graph is **Eulerian** if it contains an Euler tour.

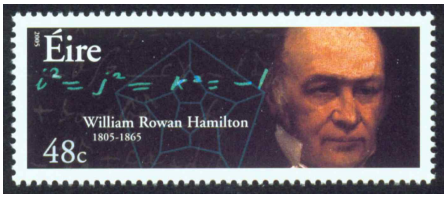
verification process: check if graph connected THEN for each vertex verify if degree of vertex is even THEN check if graph have closed walk or circuit that tranverse every edge only once
↳ polynomial ⇒ no. of edges and vertices determine its input size



Leonhard Euler (1707–1783)
https://mathshistory.st-andrews.ac.uk/Biographies/Euler/

- the set of Hamiltonian graphs
  certificate: valid Hamiltonian circuit in graph that visits each vertex only once
  - A **Hamiltonian circuit** in a graph G is a circuit which includes each vertex exactly once. (note: a circuit doesn't repeat any vertex or edge)
  - A graph is **Hamiltonian** if it contains a Hamiltonian circuit.

verification process: check if Hamiltonian circuit exists where path starts & end at same vertex and only visit each vertex only once and use each edge only once



William Rowan Hamilton (1805–1865)
https://mathshistory.st-andrews.ac.uk/Biographies/Hamilton/

↳ not efficiently done in polynomial time for arbitrary graph
↳ NP-complete problem

Our next example is fundamental in the study of graphs.
compare 2 graph to see if they are essentially the same
GRAPH ISOMORPHISM := $\{(G, H) : G \text{ is isomorphic to } H\}$
- input: graph G, graph H

G is **isomorphic** to H if there is a bijection $f : V(G) \to V(H)$ such that, for all $u, v \in V(G)$, "for all pair of vertices u and v in G, u is adjacent to v in G if and only if f(u) is adjacent to f(v) in H"

$$u \text{ is adjacent to } v \text{ in } G \quad \text{if and only if} \quad f(u) \text{ is adjacent to } f(v) \text{ in } H. \tag{25.1}$$

Such a bijection is called an **isomorphism**.
If G is isomorphic to H, we write $G \cong H$.
To say that two graphs are *isomorphic* means that the two graphs are the same apart from renaming vertices.
↳ relabel vertices of one graph so it matches the other graph structure

For example, consider the graphs G, H and J in Figure 25.1. G is a subgraph of the Melbourne rail network. H is a fragment of the British social network around the middle of the 20th century, while J is from around a century earlier.

implies that... both graph have same no. of vertices & have the same connectivity pattern
↳ may have different vertices naming

eg. graph G: A — B, A — D, B — C, D — C
graph H: I — II, I — IV, II — III, IV — III

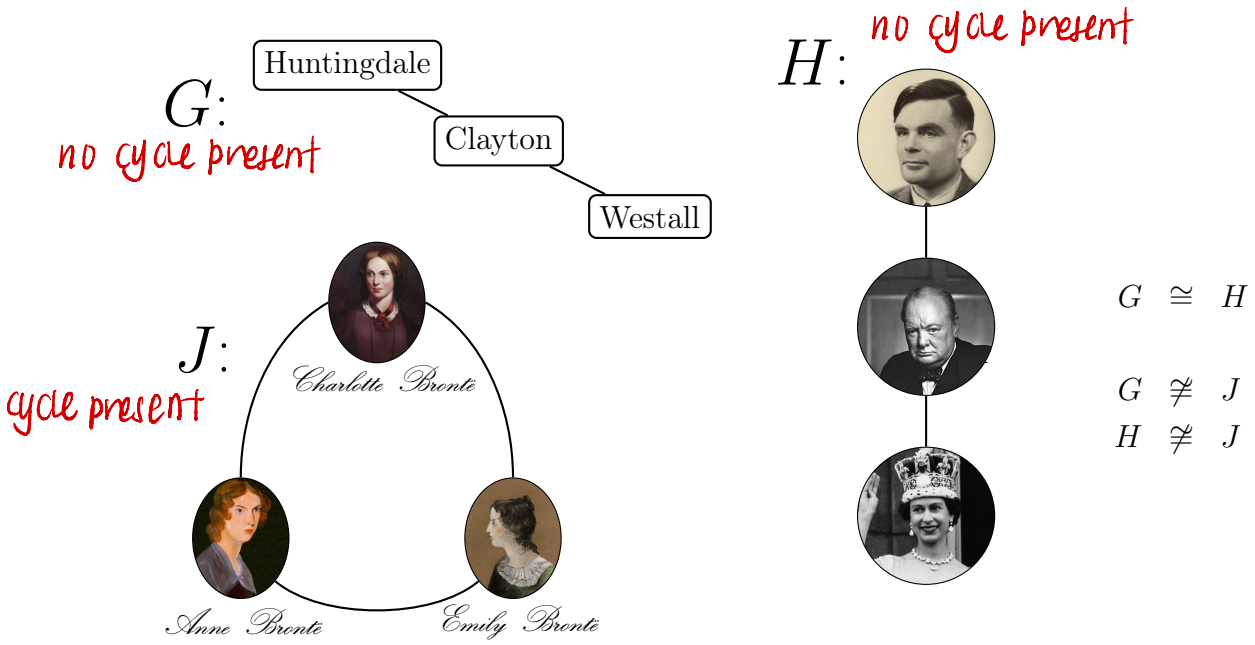


Figure 25.1: Three graphs: G, H and J.

Graphs G and H are isomorphic, and one isomorphism between them is given by the function that maps

Huntingdale	$\mapsto$	Alan Turing
Clayton	$\mapsto$	Winston Churchill
Westall	$\mapsto$	Queen Elizabeth II.

Can you find another?

But neither G nor H is isomorphic to J. Any bijection from the vertex set of G to the vertex set of J will map two nonadjacent vertices of G (Clayton and Westall) to two adjacent vertices of J, violating (25.1).

For GRAPH ISOMORPHISM, in order to verify that $G \cong H$, the natural certificate to use is a function from the vertex set of G to the vertex set of H. The verifier checks that this function is a bijection and that it preserves adjacency, i.e., condition (25.1).

We now introduce some further graph-theoretic languages in NP, chosen because of their practical importance as well as their utility in illustrating the concepts we are discussing: VERTEX COVER, INDEPENDENT SET and CLIQUE. We will revisit them later in the unit too.

For each one of these, identify a suitable certificate and a polynomial-time verifier.

A **vertex cover** in a graph $G = (V, E)$ is a set X of vertices such that every edge has at least one endpoint in X. Symbolically,

$$\forall uv \in E \quad (u \in X) \vee (v \in X)$$

VERTEX COVER

↳ vertex cover in a graph G is a set X of vertices such that every edge has at least one endpoint in X

↳ certificate: a set of valid k vertices, X, that is claimed to be a vertex cover for graph G

↳ verification process: verify set X is valid vertex cover, check every edge (u,v) in graph G have at least one of u or v is in X

↳ polynomial time ⇒ check vertices of graph G & their connections

INDEPENDANT SET

↳ independant set in graph G is a set X of vertices such that no edge have both endpoints in X

↳ certificate: a set of valid k vertices, X, that is claimed to be an independant set for the graph G

↳ verification process: verify set X is a valid independant set by checking every edge (u,v) in graph G have neither u or v in X

↳ polynomial time ⇒ check each edge of graph G & its endpoints

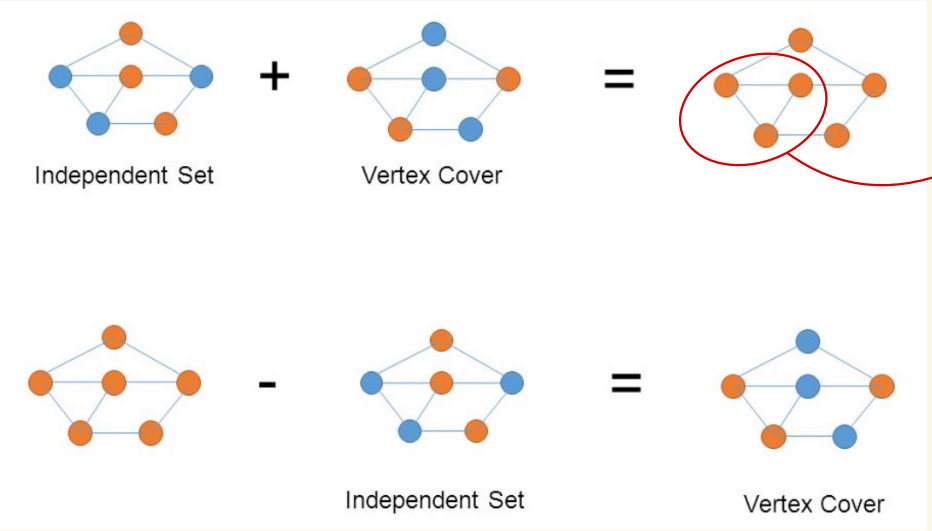
CLIQUE

↳ clique in graph G is a set X of vertices such that every pair of vertices in X are adjacent

↳ certification: a set of valid k vertices, X, that is claimed to be a clique for graph G

↳ verification process: verify set X is a valid clique by checking every pair of vertices u and v in V exist an edge (u,v) in graph G

↳ polynomial time ⇒ checking each pair of vertices in X



Relationship Between VERTEX COVER and INDEPENDENT SET:

The relationship between vertex covers and independent sets is complementary. A vertex cover is a set of vertices that covers all edges in a graph (at least one endpoint of each edge is in the cover), while an independent set is a set of vertices with no edges between them. In a graph, if X is a vertex cover, then the set of vertices not in X is an independent set, and vice versa.

↳ $\overline{\text{vertex cover}}$ = independent set

Relationship Between INDEPENDENT SET and CLIQUE:

The relationship between independent sets and cliques is also complementary. An independent set is a set of non-adjacent vertices (no edges between them), while a clique is a set of mutually adjacent vertices (an edge between every pair of vertices). In a graph, if X is an independent set, then the set of vertices not in X is a clique, and vice versa.

↳ $\overline{\text{independent set}}$ = clique

Relationship Between VERTEX COVER and CLIQUE:

The relationship between cliques and vertex covers in a graph is complementary as well, but in a slightly different way than the relationship between independent sets and cliques. A clique in a graph is a set of vertices where every pair of vertices in the set is adjacent, meaning there is an edge between every pair of vertices in the clique. A vertex cover in a graph is a set of vertices such that every edge in the graph has at least one endpoint in the vertex cover.

The complementary relationship can be described as follows:

- If you have a clique in a graph, then the set of vertices not in the clique forms a vertex cover. This is because, in a clique, there are edges between all pairs of vertices, so every edge in the graph has at least one endpoint in the complement of the clique
- If you have a vertex cover in a graph, then the set of vertices not in the vertex cover forms an independent set, where no two vertices in the set are adjacent. This is because if you have a vertex cover that covers all edges, the complement of that cover must not have edges between its vertices, creating an independent set

Summary
Vertex covers and **independent sets** have a complementary relationship, and **independent sets** and **cliques** also have a complementary relationship
Cliques and **vertex covers** have a complementary relationship in terms of graph theory: - A **clique** corresponds to a set of mutually adjacent vertices - A **vertex cover** corresponds to a set of vertices that cover all edges in the graph

↳ any pair of vertices within the clique is connected to each other

- clique is a set of mutually adjacent vertices
- vertex cover is a set of vertices that covers all edge in a graph

not in the clique = complement of clique = $\overline{\text{clique}}$

since all vertices in the clique is connected to each other, any vertex outside of clique will have at least one edge connecting it to a vertex in the clique

further explanation →

Clarification on why every edge in the graph has at least one endpoint in the complement of the clique

In graph theory, an edge is defined by two endpoints (vertices). When you have a **clique**, it means that all the vertices within the **clique** are fully connected to each other. In other words, if you pick any pair of vertices within the **clique**, there is an edge connecting them. This is the definition of a **clique** - a set of vertices where every pair is connected by an edge.

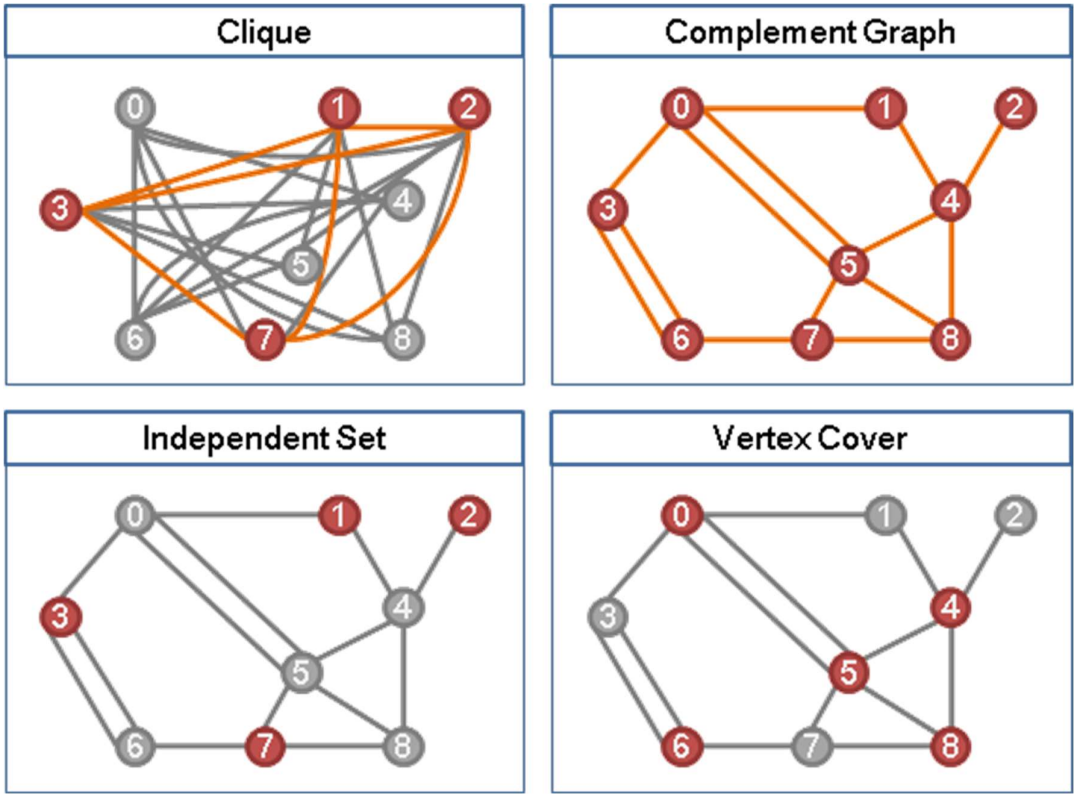
Now, consider the vertices that are ***not* in the clique**, which we refer to as the ***complement* of the clique**. Since the vertices in the **clique** are all fully connected to each other, any vertex *outside* the **clique** will have at least one edge connecting it to a vertex inside the **clique**. Why is this the case?

↗ disjointed graph

1. If a vertex outside the clique is connected to no vertices inside the clique, it would mean the clique is not maximal. In other words, you could add that vertex to the clique, and it would still satisfy the definition of a clique, contradicting the assumption that you had a clique in the first place.

2. Since the complement vertices are not in the clique, there must be at least one edge from each of these vertices that connects them to the vertices inside the clique to satisfy the definition of a clique.

So, when you look at every edge in the graph, you can be certain that at least one of its endpoints is either inside the clique or in the complement of the clique. This is why it's said that every edge in the graph has at least one endpoint in the complement of the clique.



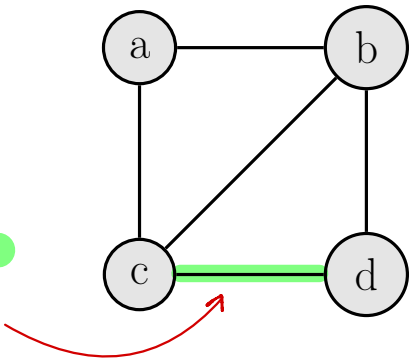
The language VERTEX COVER is defined by

$$\text{VERTEX COVER} \quad := \quad \{(G, k) : G \text{ has a vertex cover of size} \leq k\}.$$

In this graph:

{a, b, c}	is a vertex cover
{b, c}	is a vertex cover
{a, b, c, d}	is a vertex cover
{a, b}	is NOT a vertex cover

↳ edge {c,d} is not covered by vertex cover



VERTEX COVER has many applications. For example, suppose you have a network of devices, which can be represented as a graph where the vertices represent devices and the edges represent links between devices. Suppose you want to install monitoring tools on a subset of your devices in order to check that the links between devices are operating correctly. In order to minimise cost, you want to install these tools on as few devices as possible, subject to the constraint that every link is monitored from at least one of its two devices. You can model this as an instance of VERTEX COVER, where the elements of the vertex cover represent the devices at which the monitoring tool is installed.

An **independent set** in a graph $G = (V, E)$ is a set X of vertices such that no edge has both endpoints in X. Symbolically,

$$\forall uv \in E \quad (u \notin X) \vee (v \notin X)$$

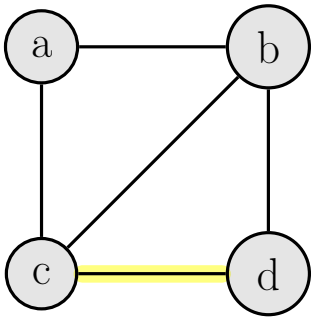
The language INDEPENDENT SET is defined by

$$\text{INDEPENDENT SET} \quad := \quad \{(G, k) : G \text{ has an independent set of size} \geq k\}.$$

In this graph:

∅	is an independent set
{b}	is an independent set
{a, d}	is an independent set
{c, d}	is NOT an independent set

↳ {c,d} connected with (c,d)



INDEPENDENT SET has applications in facility location. Suppose you have a network of possible locations for facilities of some type. In some situations, you may want to avoid having facilities that are too close together. This may be because there is a risk to these facilities such that, if one facility is adversely affected, then any neighbouring facilities are

independent set

↳ no adjacent vertices independent to each other

at risk too. You may want to determine how many facilities you can have without locating any of them close to each other, to minimise risk. This can be modelled as an instance of INDEPENDENT SET.

What is the relationship between vertex covers and independent sets?

A **clique** in a graph $G = (V, E)$ is a set X of vertices such that every pair of vertices in X are adjacent. Symbolically,

$$\forall u \in X \; \forall v \in X : \; uv \in E$$

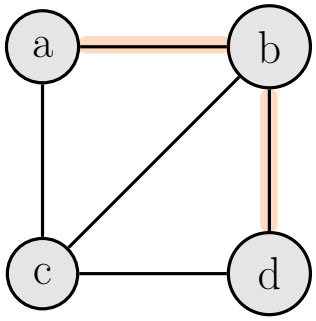
The language CLIQUE is defined by

$$\text{CLIQUE} \quad := \quad \{(G, k) : G \text{ has a clique of size} \geq k\}.$$

In this graph:

$\emptyset$	is a clique
{a}	is a clique
{a, b}	is a clique
{a, b, c}	is a clique
{a, b, d}	is NOT a clique

↳ if there a vertex from a to a then its a clique



Many applications of CLIQUE involve detecting groups of objects that are similar or close in some sense. For example, cliques have been used in social network analysis to model groups of people who all know each other.

What is the relationship between independent sets and cliques?

25.5 P and NP

We now look at the relationship between NP and other classes we have met. Firstly, we show that NP contains all of P.

Theorem 57
P $\subseteq$ NP.

Proof.
For any L in P, there is a polynomial-time decider for L.
We can turn this decider into a verifier, which works as follows.

Input: string x

Certificate: any other string y. // *. . . but ignore it!*

Run the L-decider on x.

If the decider accepts: Accept.

If the decider rejects: Reject.

If $x \in L$ then this algorithm accepts x regardless of the certificate y, since it ignores y. So it accepts for every possible certificate y. So it is certainly the case that there is at least one certificate that makes the algorithm accept.

If $x \notin L$ then the algorithm rejects x regardless of the certificate y, since it ignores y. So it rejects for every possible certificate y.

So it satisfies the conditions for being a verifier for L. (Check the definition of *verifier* that we gave earlier, to satisfy yourself of this.)

The algorithm also runs in polynomial time. The time it takes is essentially just the time taken to run the L-decider on x, and the L-decider runs in polynomial time, so our verifier runs in polynomial time too.

So we have a polynomial-time verifier for L. So, by definition, $L \in$ NP. □

Whenever we consider one class of languages that is a subset of another class of languages, it is natural to ask if it is a *proper* subset. In this case, the answer is unknown!

Open problem. Does P = NP ?

This is the biggest open problem in Computer Science and one of the biggest open problems in Mathematics. Whoever solves this problem will win fame and glory, plus wealth in the form of a Millennium Prize ($US 1 million) from the Clay Mathematics Institute (`http://www.claymath.org/millennium-problems`).

The problem has been worked on by many researchers. This work has helped build a rich and deep theory of these classes and some related complexity classes. It has shed light on the rich variety of languages that belong to these classes and made links with other models of computation. But the problem does not seem close to being solved. It needs a new approach, quite possibly from a young researcher who is freer, in their thinking, from current ways of approaching the problem. So, bright young researchers with fresh ideas are to be encouraged to learn about the problem and see what they can contribute. But it should also be pointed out that care is required, along with some humility. Many false solutions have appeared, and continue to appear, and some of these "solvers" have not been receptive to feedback and corrections. Realistically, the problem is most unlikely be solved by some routine application of existing techniques. Such approaches have been tried and tried again, unsuccessfully, and in fact there are already theorems that tell us that certain types of conventional approaches cannot resolve the P-versus-NP question.

The prevailing belief is that P is indeed a proper subclass of NP.

Conjecture. P ≠ NP.

In a 2019 survey of complexity theorists, 89% of respondents stated that they believed P $\neq$ NP while 12% believed P = NP (William I. Gasarch, Guest Column: The Third P=?NP Poll (SIGACT News Complexity Theory Column 100), *ACM SIGACT News* **50** (1) (March 2019) pp. 38–59, https://doi.org/10.1145/3319627.3319636). That survey was the third in a series that has been done every decade or so, and the story from these surveys is that the general belief that P $\neq$ NP has strengthened over time.

The case for P $\neq$ NP

One reason for believing this conjecture is the long experience in the research community that those languages in NP that are not known to be in P do seem to be fundamentally different, in their algorithmic character, to languages in P. They are generally less amenable to elegant mathematical theories. They are also very diverse. We have seen this, to some extent, in the languages in NP that we have met in this lecture. Some are about graphs, others are about numbers, others are about logical expressions.

We shall see in Lectures 27–30 that there are languages in NP called *NP-complete* languages which are (in a sense to be described) as hard as anything in NP, so that, if P $\neq$ NP, then *all* these NP-complete languages are not in P. Conversely, if *any one* of these languages has a polynomial-time algorithm, then *every* language in P has a polynomial-time algorithm, and P = NP.

The NP-complete languages are a rich and varied class, drawn from many different domains. They have resisted algorithmic attacks based on methods from graph theory, number theory, logic, and many other areas. They are of great practical importance, so there is no lack of motivation and effort to solve them. This sustained failure to find polynomial-time deciders for them, in spite of the range of methods used and the huge effort expended, gives the *impression* that there is something fundamentally intractable about them.

For further support, we quote a first-year textbook coauthored by Les Goldschlager (1951–2022), who did his PhD in complexity theory and became a Professor of Computer Science at Monash.

> "Another reason for believing that NP-complete problems are infeasible is that we experience the same dichotomy in real life between problems which are easy to solve and those which are hard to solve but whose solution appears obvious once it is found."
>
> L. Goldschlager and A. Lister, *Computer Science: A Modern Introduction*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

The case for P = NP

I think the possibility that P = NP deserves to be taken more seriously. Algorithms are remarkably difficult to analyse, in general. To illustrate this point, consider the following famous algorithm, known as the **Collatz algorithm** or the **3x+1 algorithm**, which takes any positive integer as input.

1. Input: $x \in \mathbb{N}$
2. while $x \neq 1$

3. {
4. if x is even then
5. $x := x/2$
6. else // x is odd
7. $x := 3 * x + 1$.
8. }
9. Accept.

The **Collatz language** is the set of all positive integers that are accepted by this algorithm. Currently, it is believed that the Collatz language consists of *all* positive integers, but no-one has been able to prove this; it seems well beyond current mathematical techniques. It's a very simple algorithm indeed — far simpler than many you are called upon to analyse in your computer science studies — yet we seem unable to analyse it properly: from the viewpoint of language classes, all we know is that the Collatz language is recursively enumerable (why?); we do not even know that it is decidable, yet we *believe* it is just $\mathbb{N}$ which, as a language, is computationally trivial! This is an enormous gap, spanning the entire range of language classes we have considered, from r.e. all the way down to one of the simplest regular languages. This is a very dramatic illustration of the difficulty of analysing algorithms.

If such a simple algorithm is so hard to analyse properly, what are our prospects for analysing more complex algorithms? When we write algorithms, we write them so that they are readable, comprehensible, analysable and implementable. But there are many algorithms (and Turing machines) "out there" that might be unusable by humans because they are incomprehensible and unanalysable. Some such algorithms might, conceivably, solve NP-complete problems in polynomial time. But they might be useless to us. We might not be able to prove that they solve our favourite NP-complete problem. We might not be able to prove that they run in polynomial time. We might not know them for what they are, even if we were staring them in the face.

It is worth quoting Donald Knuth on this point.

> "The author of this book ... suspects that [polynomial-time algorithms for NP-complete languages] do exist, yet that they're unknowable. Almost all polynomial-time algorithms are so complicated that they lie beyond human comprehension, and could never be programmed for an actual computer in the real world. Existence is different from embodiment."
>
> Donald E. Knuth, *The Art of Computer Programming. Volume 4B. Combinatorial Algorithms, Part 2*, Addison-Wesley, Boston, 2022, p. 185 (footnote).

The case for P $\stackrel{?}{\neq}$ NP

A small minority of researchers believe that the P-versus-NP question might be independent of the axioms of mathematics. If this is the case, then neither P = NP nor P $\neq$ NP could be proved. Maybe, in the future, some new axiom might be introduced and accepted

as part of the foundations of mathematics, and maybe the enlarged set of axioms will then be strong enough to prove P = NP or P ≠ NP. Maybe one of these statements — P = NP or P ≠ NP — will itself become accepted as a new axiom! Or maybe two separate branches of mathematics will develop, one using the usual axioms plus P = NP, the other using the usual axioms plus P ≠ NP. The research community is doing something a little like this already, as an interim measure, by prefacing many theorem statements with the required assumptions: "If P = NP ..." or "If P ≠ NP ...".

The case for a different question

In asking whether or not P equals NP, are we asking the right question? An NP-complete problem is not really "solved" by an algorithm for which we cannot *prove* that it solves the problem, or that it runs in polynomial time (or both), or which is in some sense too complicated to be implemented in a readable, comprehensible, maintainable program.

If some "unknowable" polynomial-time algorithms solves an NP-complete problem, then this may tell us that P-versus-NP is not the right way to distinguish between efficient decidability and efficient verifiability. We might need to develop a new notion of efficiency that covers more than just raw time complexity and includes some formalisation of our goals of provability and human implementability as well.

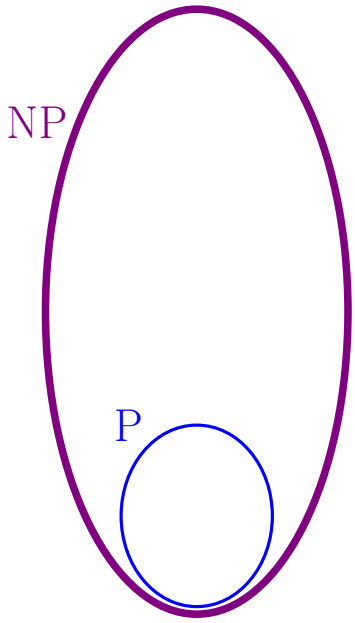
Even if the P-versus-NP question is resolved in a more conventional way, it is arguable that a new question is needed.

> "The very best of problems, if it happens to fall to a boring technique, was the wrong problem. Suppose somebody were to prove P ≠ NP then this would be of great personal and dramatic interest, in view of the people that have tried to work on it. But if it were to be established by a quite traditional, ordinary argument, which it just happened that nobody had thought of, then we would see in retrospect that it was the wrong problem."
>
> Mike Paterson, on pp. 184–185 of the Panel Discussion at the end of: R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations (Proceedings of a Symposium held March 20–22, 1972, IBM Thomas J. Watson Research Center, Yorktown Heights, New York)*, Plenum Press, New York, 1972.

25.6 Language classes

It is time for another Venn diagram. In fact, we give two of them, the first assuming that P ≠ NP and the second assuming that P = NP. On the right of the diagrams we list some of the languages belonging to the classes.

If P $\neq$ NP:

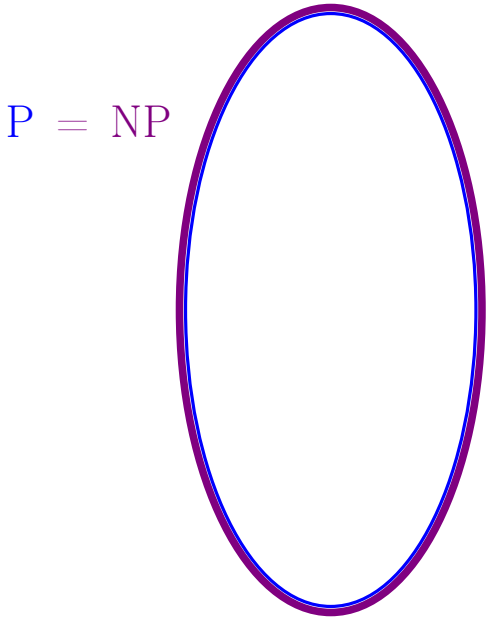


In NP, not known to be in P:
SATISFIABILITY, 3-SAT,
HAMILTONIAN CIRCUIT,
3-COLOURABILITY,
VERTEX COVER,
INDEPENDENT SET, ...

GRAPH ISOMORPHISM,
INTEGER FACTORISATION, ...

In P:
2-SAT,
EULERIAN CIRCUIT,
2-COLOURABILITY,
CONNECTED GRAPHS,
SHORTEST PATH,
PRIMES,
Invertible matrices,
...,
All Context-Free Languages,
All Regular Languages.

If P = NP:



SATISFIABILITY, 3-SAT,
HAMILTONIAN CIRCUIT,
3-COLOURABILITY,
VERTEX COVER,
INDEPENDENT SET, ...

GRAPH ISOMORPHISM,
INTEGER FACTORISATION, ...

2-SAT,
EULERIAN CIRCUIT,
2-COLOURABILITY,
CONNECTED GRAPHS,
SHORTEST PATH,
PRIMES,
Invertible matrices,
...,
All Context-Free Languages,
All Regular Languages.

25.7 NP and decidability

Having considered the relationship between NP and its subclass P, we must also consider the relationship between NP and the large class of decidable languages. Unlike the case of P and NP, the definitions of NP and decidability do not immediately imply a subset

relation one way or the other. We are comparing the power of *polynomial*-time verifiers with *unlimited*-time deciders. The former has less time but more information (the certificate).

Fortunately, we can show that all languages in NP are decidable, and in fact that they are decidable in exponential time.

Theorem 58

Any language in NP can be decided in time $O(2^{n^K})$ for some constant K.

Proof.

Let L be any language in NP. It has a polynomial-time verifier, V (by the definition of NP).

We will construct from V a *decider* for L that does an exhaustive search of all possible certificates, checking each certificate with V to see if one of the certificates gets the input accepted.

This decider for L works as follows.

> Input: x
>
> For each certificate y:
>
> > Call the verifier V on input x with certificate y.
> >
> > If V accepts, then Accept, else continue.
>
> Reject. // *. . . since, if we reach here, no certificate leads to acceptance.*

This decider:

- accepts x if the verifier accepts for some y;
- rejects x if the verifier rejects for every y.

So this decider is a decider for L (using the definition of a verifier).

It remains to consider the time complexity of our decider.

Since the verifier runs in polynomial-time, it has time complexity $O(n^k)$ for some constant k. The decider runs the verifier once for each certificate. So the decider's time complexity is $O((\#\text{ certificates}) \cdot n^k)$.

We therefore need an upper bound on the number of certificates. At first sight, it may seem that there are infinitely many certificates, since we have allowed certificates to be any strings at all and have not put any explicit upper bound on their length.

BUT in t steps, a Turing machine can examine at most t symbols in the certificate.

Our verifier has time complexity $O(n^k)$, which is $\leq c \cdot n^k$ (for sufficiently large n).

So this verifier sees $\leq c \cdot n^k$ symbols in the certificate. Any symbols beyond that are ignored.

Assuming our usual alphabet $\{\mathtt{a}, \mathtt{b}\}$, the number of certificates *that need to be checked* is $\leq 2^{cn^k}$.

So, the decider's total time complexity is $O(2^{cn^k} n^k)$.

This is dominated by the exponential part. In fact you can find a constant K a bit larger than k such that the time complexity is $O(2^{n^K})$.[1]

So any language in NP can be decided in exponential time. □

25.8 Nondeterministic Turing machines

All Turing machines so far have been **deterministic**, i.e., for each state and symbol, there is at most one transition. So, for each state and for each symbol, the next action is completely determined: there is a specific next state, new symbol and direction, or if no transition is specified, the TM crashes. In fact, the entire computation is completely determined by the input.

In a **nondeterministic Turing machine** (NDTM or NTM), for a given state and symbol, there may be more than one possible transition. (This was briefly mentioned late in Lecture 18.) The computation an NDTM carries out is not necessarily completely determined by the input; a single input may lead to many possible computations.

A deterministic TM (DTM) is also a NDTM. DTMs are special cases of NDTMs in which ambiguity never arises.

The **language accepted** by a NDTM M is the set of input strings for which *some* computation leads to an Accept state.

A NDTM M is a **nondeterministic decider** for a language L if

- M halts on all inputs, and
- the language accepted by M is L.

A **polynomial-time NDTM** is a NDTM with time complexity $O(n^k)$, for some fixed k. As usual, n is the length of input string. Time complexity is maximum time taken over all inputs of length n (as usual) *and* all possible computations for each of those inputs.

Theorem 59

L is in NP if and only if some polynomial-time NDTM is a nondeterministic decider for L.

↳ L contains problems that can be verified in polynomial time if there exist NDTM that can guess and check multiple possibilities simultaneously in polynomial time.

Proof. (outline)

↳ non-deterministically guess and decide membership

($\Longrightarrow$)

Suppose L has a verifier with time complexity $\leq c\,n^k$.

Construct a NDTM M as follows.

On input x, M generates a string y of length $c\,n^k$, nondeterministically, and then just executes the verifier on x, y.

---

[1] One way to help see this: $2^{cn^k} n^k = 2^{cn^k} \cdot 2^{\log_2(n^k)} = 2^{cn^k + \log_2(n^k)} = 2^{cn^k + k\log_2 n} < 2^{cn^k + kn} \leq 2^{dn^k + dn} \leq 2^{2dn^k} \leq 2^{n^K}$, with the last inequality holding for all sufficiently large n, where $K > k \geq 1$ and $d \geq \max\{c, k\}$ are constants.

($\Longleftarrow$)

Let M be a polynomial-time NDTM that decides L.

Set up a way of encoding, as a string, the sequence of choices made at the nondeterministic steps of a computation.

Use this string as a certificate ...

□

3-SAT belongs to NP so complement of 3-SAT also belongs to NP

NP stands for Nondeterministic Polynomial time.

We have now seen nondeterminism in four different computational contexts. We summarise the effect of adding nondeterminism, in terms of language-recognition power, for each of these contexts. For each computational model, the entry in the right column answers the question: Can nondeterministic machines recognise more languages than deterministic ones?

computational model	nondeterministic > deterministic?
Finite Automata (FA / NFA)	No
Pushdown Automata (DPDA / PDA)	Yes
Polynomial-time TMs (p-time deciders / p-time verifiers)	Unknown
Turing machines (TM / NDTM)	No

There is no clear pattern here. There are many other computational models and they too vary in whether or not nondeterminism enables recognition of more languages.

Revision

you can define co-NP as the class of decision problems for which the answer is 'no' and its complement is in NP. By analogy with co-re, co-NP represents problems whose complement can be verified

Things to think about:

- Does $\overline{\text{2-SAT}}$, the complement of 2-SAT, belong to P?
  yes, 2-SAT is in P so its complement is also in P
- Does $\overline{\text{3-SAT}}$, the complement of 3-SAT, belong to NP?
- Can you define co-NP, by analogy with co-r.e.?
  yes, you can define co-NP by analogy with co-re
- What is the class of languages that have a verifier?
  (...with no requirement for it to be *polynomial time*)
  recursively enumerable (RE)
  → includes all problems for which proposed solution can be checked by Turing machine

** co-NP = complement of NP

Reading:

- Sipser, sections 7.2–7.3. even if verification process is not terminated for some input (not decidable)
- M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., San Francisco, 1979: §2.3, §2.4.

co-NP is the class of decision problems for which the answer is 'no' and proposed solution claim 'yes' where proposed solution can be verified in polynomial time

whether certain problems belong to co-NP depends on their ability to have efficiently checkable 'no' answers

Revision Questions

1. **Does 2-SAT, the complement of 2-SAT, belong to P?**

- 2-SAT is a special case of the Boolean satisfiability problem where each clause has exactly 2 literals. It's known that 2-SAT is in P and can be solved in polynomial time.

- The complement of 2-SAT is also in P because if a problem is in P, its complement is also in P. This is a property known as closure under complement.

→ 3-SAT belongs to NP so complement of 3-SAT also belongs to NP

2. **Does 3-SAT, the complement of 3-SAT, belong to NP?**

- 3-SAT is the classic NP-complete problem, which means it's NP-hard and in NP. However, the complement of an NP-complete problem is not necessarily in NP. It could be in co-NP, but we don't know for sure.

- Whether 3-SAT's complement is in NP or co-NP is an open question and is part of the famous P vs. NP problem, which remains unresolved.

3. **Can you define co-NP, by analogy with co-r.e.?**

- Co-NP is the class of decision problems for which the answer is "no." In other words, a language L is in co-NP if and only if its complement (the set of strings not in L) is in NP.

- This is similar to the definition of co-recursively enumerable (co-r.e.) in the context of recursive enumerability. If a problem is in co-NP, it means that a proposed solution can be efficiently verified.

4. **What is the class of languages that have a verifier (with no requirement for it to be polynomial time)?**

- The class of languages for which there exists a verifier (a machine that can efficiently check the correctness of a proposed solution) without the requirement for it to be polynomial time is known as the class **RE** (Recursive Enumerable).

- RE includes all problems for which a proposed solution can be checked by a Turing machine, even if the verification process may not terminate for some inputs. These problems may not be decidable (i.e., they may not always halt), but they are at least recursively enumerable.

N, NP and NP-COMPLETE

N (Nondeterministic Polynomial Time):

- N represents the class of decision problems that can be solved in polynomial time on a nondeterministic Turing machine.
- In simpler terms, problems in class N can be verified in polynomial time, but we don't necessarily know how to find the solution efficiently.
- Eg. We're checking if a number is prime in the code below. Verifying that a number is prime can be done in polynomial time, so it's in class N.

```
def isPrime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

NP (Nondeterministic Polynomial Time):

- NP represents the class of decision problems for which a proposed solution can be verified in polynomial time on a deterministic Turing machine.
- While NP problems can't necessarily be solved in polynomial time, if you give a proposed solution, you can efficiently check if it's correct.
- If we were to check if a subset of numbers sums up to a target value. Verifying a solution (checking if a subset sums to the target) can be done in polynomial time, so it's in class NP.

NP-COMPLETE:

- NP-COMPLETE represents a special class of problems within NP that are as hard as the hardest problems in NP.
- If you can find an efficient algorithm to solve any NP-COMPLETE problem, you can solve all problems in NP efficiently.
- Eg. Traveling Salesman Problem (TSP). It's a combinatorial optimization problem where you want to find the shortest possible route that visits a set of cities exactly once and returns to the starting city. There's no known polynomial-time algorithm to solve it optimally.

```
# Example of a brute-force solution for the TSP
def traveling_salesman(graph, start):
    # Generate all possible permutations of cities
    permutations = itertools.permutations(graph.keys())
    min_distance = float('inf')
    best_path = None

    for perm in permutations:
        path = list(perm) + [start]
        distance = calculate_total_distance(path, graph)
        if distance < min_distance:
            min_distance = distance
            best_path = path

    return best_path, min_distance
```

- This code demonstrates a brute-force approach to solving the TSP, but it's not efficient for larger instances, which is a characteristic of NP-COMPLETE problems.

Summary

- N represents problems that can be solved and verified in polynomial time.
- NP represents problems for which proposed solutions can be verified in polynomial time.
- NP-COMPLETE are believed to be not solvable in polynomial time.

NP vs CO-NP

CO-NP:

Problem: Composite Numbers

Given an integer n, determine whether it's a composite number (not prime). In other words, the problem is to verify that the answer is "no" and provide evidence that n has divisors other than 1 and itself.

Verification for "No" Answer:

To verify that an integer n is not prime (i.e., the answer is "no"), you can check if it has divisors other than 1 and itself. If you find such divisors, you can efficiently conclude that n is composite.

Example:

Let's say n = 15. To verify that it's not prime, you can check if it has divisors other than 1 and 15. In this case, you can easily find divisors, such as 3 and 5. This verification process is efficient and confirms that the answer is "no" (15 is not prime).

This problem is in co-NP because it involves verifying the "no" answer (n is not prime) efficiently. If someone claims that an integer is not prime (answer "no"), you can efficiently check and confirm their claim by identifying divisors.

NP:

Problem: Hamiltonian Path

Given a directed or undirected graph G, is there a path that visits every vertex exactly once, and returns to the starting vertex? This is called a Hamiltonian Path.

Verification for "Yes" Answer:

If someone claims that a Hamiltonian path exists in a given graph, you can verify their claim efficiently by checking that the path they provide indeed visits each vertex once and returns to the starting vertex. This verification can be done in polynomial time.

Example:

Consider an undirected graph G with the following vertices and edges:

Vertices: A, B, C, D

Edges: (A, B), (B, C), (C, D), (D, A)

Claim: There is a Hamiltonian path in the graph.

You can verify this claim by examining a path that starts at one vertex (e.g., A), follows the edges (A -> B -> C -> D), and returns to the starting vertex (A). This path satisfies the Hamiltonian path condition, and you can verify it efficiently.

Since the verification of a "yes" answer (the existence of a Hamiltonian path) can be done in polynomial time, the Hamiltonian Path problem is in NP.

In co-NP, you would typically have problems where you efficiently verify a "no" answer. Problems like Prime Factorization is an example of a problem in NP where you efficiently verify a "yes" answer. If you find a number's prime factors, you can easily verify their correctness by multiplying them.

Lecture 26

Polynomial-time reductions

In Lecture 21 we studied mapping reductions and used them to establish computational relationships between problems. We will now do the same kind of thing for *polynomial-time* computation.

26.1 Definitions

A **polynomial-time reduction** from K to L is a *polynomial-time mapping reduction* from K to L.

So, it's a *polynomial-time* computable function

$$f : \Sigma^* \to \Sigma^*$$

such that, for all $x \in \Sigma^*$,

$$x \in K \quad \text{if and only if} \quad f(x) \in L$$

Polynomial-time reductions are also called:

- polynomial-time mapping reductions
- polynomial-time many-one reductions
- polynomial transformations
- Karp reductions

If there is a polynomial-time reduction from K to L, then we write $K \leq_P L$.

26.2 Examples of polynomial-time reductions

One place we can look for examples is mapping reductions!

Which of the **mapping reductions** in Lecture 21 are **polynomial-time**?

	Yes	No
EQUAL $\longrightarrow$ HALF-AND-HALF	☑	☐
HALF-AND-HALF $\longrightarrow$ PARENTHESES	☑	☐
FA-Empty $\longrightarrow$ No-Digraph-Path	☑	☐
RegExpEquiv $\longrightarrow$ FA-Empty	☐	☑

Try to answer each of these before reading on.

Now let's look at each of them in turn.

EQUAL $\longrightarrow$ HALF-AND-HALF

This mapping reduction just requires sorting of the input string. There are many sorting algorithms, and many of them deal with significantly more complex situations than we face here. The comparisons we need to do, while sorting our input string, are only between the letters `a` and `b`, with `a` $<$ `b`. The standard Bubble Sort algorithm works by repeated comparisons between consecutive members of a sequence. This is readily adapted to our situation here: we just do repeated comparisons between consecutive letters on the Turing machine's tape. Bubble Sort requires $O(n^2)$ comparisons, where n is the number of terms of the sequence of objects being sorted. When we adapt this to a Turing machine sorting our input string, each comparison requires a constant number of Turing machine steps, in order to check two adjacent letters and swap them if necessary. So we can program a Turing machine to do Bubble Sort on the input string, and it will take $O(n^2)$ steps, which is polynomial time.

If you have studied sorting algorithms, you will know that there are many faster algorithms than Bubble Sort, with the fastest doing $O(n \log n)$ comparisons. But, if all we are doing is working out whether or not a task can be done in polynomial time, we don't usually need to consider the very fastest possible algorithms. In fact, the Bubble Sort algorithm is among the easiest sorting algorithms to program and analyse, and is readily implemented in a Turing machine for the situation we have here. So it suits our present purpose, of showing that this mapping reduction is polynomial time, very well.

HALF-AND-HALF $\longrightarrow$ PARENTHESES

For this mapping reduction we need to work our way rightwards along the tape, doing the required letter replacements as we go. We also need to detect if we ever meet the substring `ba`, but this can be done using appropriate states.

- If we do detect `ba`, then we return to the start, erasing letters as we go, and replace the first letter as required. (For this to be possible, we need to have marked the first letter appropriately in the first step of our computation, as we have often done with Turing machine computations.) This ensures that our single-letter output string has blanks after it, so that it is indeed the output of the Turing machine (according to our definition of Turing machine output, which does not exclude the possibility of non-blanks beyond the first blank, with any such non-blanks not being part of the output string). In the worst case, we go all the way along the string until we find `ba` at the very end, then go all the way back, erasing as we go. The number of Turing machine steps should be $\leq 2n + 1$ (including, at the every end, a transition into the Accept state which takes a single rightward step on the tape after replacing the first letter).

- If we never detect `ba`, then we just go all the way along the input string, doing the specified replacements as we go, until we reach the first blank. But we can't accept straight away, as we will have marked the first letter with a different symbol (such as `A` or `B`), because at the very start we did not know whether we would find `ba` or not. So we have to go all the way back to the start to do the appropriate replacement for the first letter, and then Accept. The number of Turing machine steps is exactly $2n + 1$.

So, in any case, the time complexity is $\leq 2n + 1$, which is $O(n)$, so this mapping reduction is polynomial-time computable.

FA-Empty $\longrightarrow$ No-Digraph-Path

The essence of this computation is the conversion of a Finite Automaton into a directed graph, with identification of the start and destination vertices. The details depend on the way we represent Finite Automata and directed graphs as strings. But, under most natural encoding schemes, the changes required are very local, and straightforward. Suppose we have a transition from state v to state w labelled by x, where x is a letter. (You could, if you wished, imagine an encoding scheme for this, like a simplified version of the way we represented Turing machine transitions in §19.1.) To represent this as a directed edge, we just "throw away" the information about the letter and just represent the ordered pair (v, w). There are many ways to do this. An algorithm could be devised that does a constant (or even linear, if we did not want to be too fussy) amount of work per FA transition. The main loop that does this would dominate the time complexity, although there are other small tasks too, like the extra edges going out of all the Final States and the identification of the special vertices s and t. The time complexity should be $O(n^2)$ or better, and certainly polynomial time.

RegExpEquiv $\longrightarrow$ FA-Empty

This algorithm includes conversion of regular expressions to NFAs and, at some stage, the conversion of at least one NFA to an FA. The NFA-to-FA conversion can take exponential time, because an NFA of k states may be converted to an FA of nearly 2^k states (in the worst case). So this mapping reduction is not polynomial-time computable.

We now give mapping reductions involving some of the languages of graphs that we introduced in the previous lecture.

Theorem 60

INDEPENDENT SET $\leq_P$ CLIQUE.

Proof.

Our mapping reduction will use the following construction.

The **complement** $\overline{G}$ of a graph G is defined to be the graph obtained from G by flipping the status of each pair of vertices, so that every vertex pair that is adjacent in G becomes non-adjacent in $\overline{G}$ and every pair of vertices that are non-adjacent in G becomes adjacent in $\overline{G}$.

Now, based on the definitions of independent sets and cliques (Lecture 25, p. 274), we can see that ndependent sets in G correspond to cliques in $\overline{G}$.

So, G has an independent set of size $\geq k$ if and only if $\overline{G}$ has a clique of size $\geq k$.

So:

$$(G, k) \in \text{INDEPENDENT SET} \quad \text{if and only if} \quad (\overline{G}, k) \in \text{CLIQUE}.$$

Construction of $(\overline{G}, k)$ from (G, k) is polynomial time. (The core of the reduction is to loop over all vertex pairs of G, doing a small amount of work for each. The number of vertex pairs is $\binom{n}{2} = O(n^2)$, where n is the number of vertices of G.)

So the function

$$(G, k) \mapsto (\overline{G}, k)$$

is a polynomial-time reduction from INDEPENDENT SET to CLIQUE. □

Theorem 61

VERTEX COVER $\leq_P$ INDEPENDENT SET.

Proof.

If G is a graph and $X \subseteq V(G)$, then:

X is a vertex cover of G if and only if $V(G) \setminus X$ is an independent set of G.

So:

$$(G, k) \in \text{VERTEX COVER} \quad \text{if and only if} \quad (G, n-k) \in \text{INDEPENDENT SET}.$$

The construction is polynomial time, since all we have to do is one subtraction.

So the function

$$(G, k) \mapsto (G, n-k)$$

is a polynomial-time reduction from VERTEX COVER to INDEPENDENT SET. □

At this point, consider whether you can reduce the other way in either of these cases. Is there a polynomial-time reduction from CLIQUE to INDEPENDENT SET? Is there one from INDEPENDENT SET to VERTEX COVER?

Also, we have said nothing about the computational relationship between CLIQUE and VERTEX COVER. Can you do a polynomial-time reduction from one to the other, and if so, can you do it in both directions?

The following language is reminiscent of GRAPH ISOMORPHISM which we met in Lecture 25 (p. 271).

$$\text{SUBGRAPH ISOMORPHISM} := \{(G, H) : G \text{ is isomorphic to a } \textit{subgraph} \text{ of } H\}.$$

SUBGRAPH ISOMORPHISM is quite different, computationally, to GRAPH ISOMORPHISM, because of the many different ways in which the vertex set of G could map *into* the (potentially much larger) vertex set of H (whereas GRAPH ISOMORPHISM requires a bijection between the two vertex sets).

Theorem 62

$$\text{GRAPH ISOMORPHISM} \leq_P \text{SUBGRAPH ISOMORPHISM}$$

Proof.

When G and H have the same number of vertices, the only way G can be isomorphic to a *subgraph* of H is for it to be isomorphic to H itself. So, for our reduction, we would like to use the identity map,

$$(G, H) \mapsto (G, H).$$

As inputs to GRAPH ISOMORPHISM, these graphs G and H are not required to have the same number of vertices. So this mapping is not the full polynomial-time mapping reduction we need. But, if G and H have different numbers of vertices, then they cannot possibly be isomorphic, so we can handle them without looking at their structure in detail. We can just map (G, H) to some pair of graphs that do not belong to SUBGRAPH ISOMORPHISM.

$$(G, H) \mapsto \begin{cases} (G, H) & \text{if } |V(G)| \geq |V(H)|, \\ \left(\text{  }, \text{  } \right) & \text{if } |V(G)| < |V(H)|. \end{cases}$$

This reduction is polynomial-time because we do not have to do much more than some simple tests on the sizes of the vertex sets. □

Now we give an important polynomial-time reduction involving logical expressions.

We met 2-SAT late in Lecture 24 (p. 263). It belongs to P and therefore to NP too. We met 3-SAT in Lecture 25 (p. 271); it belongs to NP but is not known to belong to P.

Theorem 63
2-SAT $\leq_P$ 3-SAT

Proof. (outline)

Given a Boolean formula φ in CNF with 2 literals per clause, we want to transform it to another Boolean formula φ' in CNF with 3 literals/clause, such that

$$\varphi \text{ is satisfiable} \quad \text{if and only if} \quad \varphi' \text{ is satisfiable.}$$

We can do this as follows.

For each i:

Suppose the i-th clause in φ is $x \vee y$.

Create a new variable w_i which appears nowhere else.

Replace clause $x \vee y$ by two clauses:

$$(x \vee y \vee w_i) \wedge (x \vee y \vee \neg w_i)$$

For the rest of the proof, you need to show that

- this construction takes polynomial time,
- φ is satisfiable if and only if φ' is satisfiable.

□

Now we consider some languages involving numbers.

$$\text{PARTITION} := \left\{ (s_1, s_2, \ldots, s_n) \ : \ \text{for some } J \subseteq \{1, 2, \ldots, n\}, \ \sum_{i \in J} s_i = \sum_{i \in \{1,\ldots,n\} \setminus J} s_i \right\}$$

$$\text{SUBSET SUM} := \left\{ (s_1, s_2, \ldots, s_n, t) \ : \ \text{for some } J \subseteq \{1, 2, \ldots, n\}, \ \sum_{i \in J} s_i = t \right\}$$

Theorem 64
PARTITION $\leq_P$ SUBSET SUM

Proof. (outline)
Use the reduction

$$(s_1, s_2, \cdots, s_n) \;\mapsto\; (\, s_1, s_2, \ldots, s_n, (s_1 + s_2 + \cdots + s_n)/2 \,)$$

□

Can you show SUBSET SUM $\leq_P$ PARTITION ?

Here are some other polynomial-time reductions to try to find.

3-COLOURABILITY $\leq_P$ GRAPH COLOURING

where GRAPH COLOURING := $\{ (G, k) : G \text{ is } k\text{-colourable} \}$

2-COLOURABILITY $\leq_P$ 3-COLOURABILITY

HAMILTONIAN CIRCUIT $\leq_P$ HAMILTONIAN PATH

2-COLOURABILITY $\leq_P$ 2-SAT

SATISFIABILITY $\leq_P$ 3-SAT

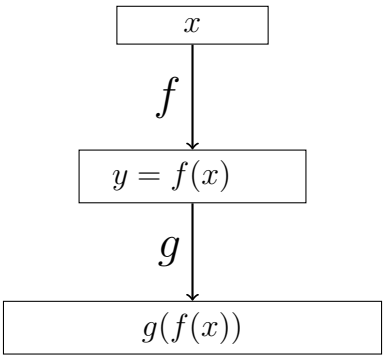
3-COLOURABILITY $\leq_P$ SATISFIABILITY

26.3 Properties

The relation $\leq_P$ is trivially reflexive: for any language L, $L \leq_P L$.
We now show that it is also transitive.

Theorem 65
If $K \leq_P L$ and $L \leq_P M$ then $K \leq_P M$.

Proof.
Let f be a polynomial-time reduction from K to L.
Let g be a polynomial-time reduction from L to M.



We've already seen (Lecture 21, proof of Theorem 42) that $g \circ f$ is a *mapping reduction* from K to M.

We just need to show that it's a *polynomial-time* mapping reduction.

Since f and g are both polynomial-time, we know that:

(i) $f(x)$ is computable in time $\leq c\,|x|^k$, for some constants c, k and all sufficiently large x.

(ii) $g(y)$ is computable in time $\leq d\,|y|^\ell$, for some constants d, ℓ and all sufficiently large y.

It follows from (i) that $|f(x)| \leq c\,|x|^k$ too, for sufficently large x, since at most one letter of output can be computed in each time-step.

It follows from (ii) that

$$\begin{array}{lll} \text{time to compute } g(f(x)) \text{ from } f(x) \text{ is} & \leq & d\,|f(x)|^\ell \\ & = & d\,(c|x|^k)^\ell \qquad \text{by above bound on } |f(x)| \\ & = & d\,c^\ell\,|x|^{k\ell} \qquad \text{for large enough } x. \end{array}$$

Therefore,

$$\begin{array}{lll} \text{time to compute } g(f(x)) & & \\ = & \text{time to compute } f(x) \text{ from } x \;+\; \text{time to compute } g(f(x)) \text{ from } f(x) & \\ \leq & c\,|x|^k \;+\; d\,c^\ell\,|x|^{k\ell} & \text{for sufficiently large } x\text{, using what we did above} \\ \leq & c'\,|x|^m & \text{for some constants } c', m \text{ and all sufficiently large } x. \end{array}$$

So $g \circ f$ is polynomial-time. □

Our next theorem gives one of the main reasons for our interest in polynomial-time reductions. It is the analogue, for the class P, of Theorem 40 (Lecture 21).

Theorem 66

If $K \leq_P L$ and L is in P, then K is in P.

1. **Polynomial-Time Reduction:**

- A polynomial-time reduction is a way to compare the computational difficulty of two problems. Specifically, it is a way to transform instances of one problem (K) into instances of another problem (L) in polynomial time.

2. **L is in P:**

- It is given that the problem L is in P. This means that there exists a deterministic Turing machine that can solve instances of L in polynomial time. In other words, L is a problem that can be efficiently solved.

3. **The Theorem Statement:**

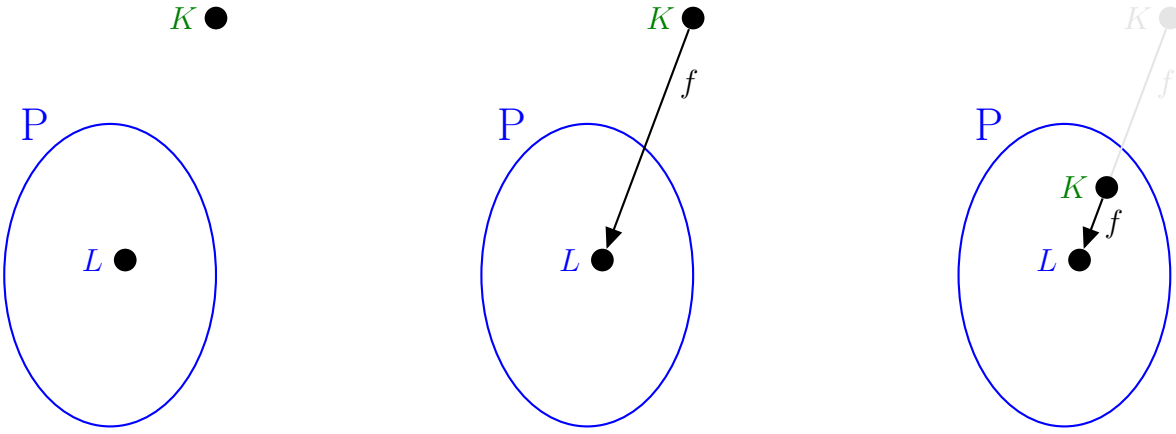
- The theorem asserts that if problem K can be polynomial-time reduced to problem L, and L is efficiently solvable (in P), then problem K is also efficiently solvable (in P).

Here's an intuitive explanation of the theorem:

- When you say problem K is polynomial-time reducible to L, it means that you can use a polynomial-time algorithm to transform instances of K into instances of L. This transformation effectively allows you to solve instances of K by solving instances of L.

- Since L is in P, there is an efficient algorithm to solve it. If you can reduce K to L efficiently, it means that you can solve K efficiently as well. This is because you can transform an instance of K into an instance of L in polynomial time and then use the efficient algorithm for L to solve it.

In other words, if you can efficiently solve L, and you can efficiently transform instances of K into instances of L, then you can efficiently solve K. This theorem highlights the idea that problems that can be reduced to efficiently solvable problems remain efficiently solvable themselves. It's a fundamental result in computational complexity theory and helps in understanding the relationships between different complexity classes.



Proof.

Let f be a polynomial-time reduction from K to L, and let D be a poly-time decider for L.

Decider for K: *(same as in Lecture 21)*

Input: x.

Compute $f(x)$.

Run the Decider for L on $f(x)$.

// *This L-Decider accepts $f(x)$ if and only if $x \in K$, since f is a mapping reduction from K to L.*

We also need to show it's polynomial time.

If f has time complexity $O(n^k)$, then the length of its output string $f(x)$ must also be $O(n^k)$, since a TM can, in t steps, output no more than t symbols.

The decider D runs in polynomial time, so suppose it has time complexity $O(n^{k'})$, where n is the size of the input to D.

If D is given $f(x)$ as input, then the time D takes on it is $O(|f(x)|^{k'})$, where $|f(x)| =$ length of string $f(x)$.

Since $|f(x)| = O(n^k)$, we find that D takes time $O(n^{kk'})$, where $n = |x|$.

Total time taken by our decider for K is:

$$\begin{aligned} \text{time taken by } f \text{ on } x \ + \ \text{time taken by } D \text{ on } f(x) \ &= \ O(n^k) + O(n^{kk'}) \\ &= \ O(n^{kk'}), \end{aligned}$$

which is polynomial time. □

Corollary 67

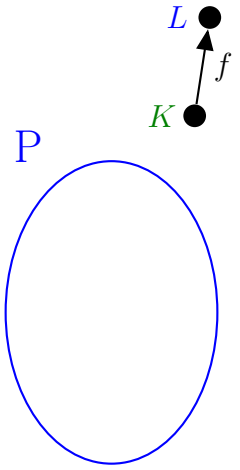
If there is a polynomial-time reduction f from K to L, then:

If K is not in P, then L is not in P.

Symbolically:

$$(K \leq_P L) \wedge (K \notin \mathrm{P}) \implies (L \notin \mathrm{P})$$

Proof. complement of previous theorem (theorem 66)
Contrapositive of previous Theorem. □



Theorems 40 and 66 have an analogue for NP.

Theorem 68
If $K \leq_P L$ and L is in NP, then K is in NP.

Writing a proof for this theorem is a good exercise.

Revision

Things to think about:

- You will have seen transformations from one problem to another before, and probably not just in this unit. Are any of them polynomial-time reductions?
- Find some of the polynomial-time reductions mentioned on p. 291.

Reading: Sipser, Section 7.4, pp. 299–303.

Lecture 27

NP-completeness

Languages in P are, in a sense, the "easiest" languages in NP, in the sense that they have "efficient" (polynomial time) deciders.

We have seen that NP contains languages pertaining to a rich variety of structures (logical expressions, graphs, number sequences, ...) and drawn from many different application domains. Although many of them are not known to be in P, their practical importance means that we cannot ignore them just because we cannot find deciders for them. We still need to understand these languages and the computational relationships between them. Polynomial-time reductions are an important tool for doing this.

We now use polynomial-time reductions to define a class of "hardest" languages in NP. This class includes many languages we have already met.

27.1 Definition

there exist a NDTM that can decide and verify a proposed solution in polynomial time

A language L is **NP-complete** if

(a) L is in NP, and

(b) every language in NP is polynomial-time reducible to L, i.e.,

$$\forall K \in \text{NP} : \; K \leq_P L.$$

For any language K in NP ($K \in NP$), there exist a polynomial-time reduction from K to L

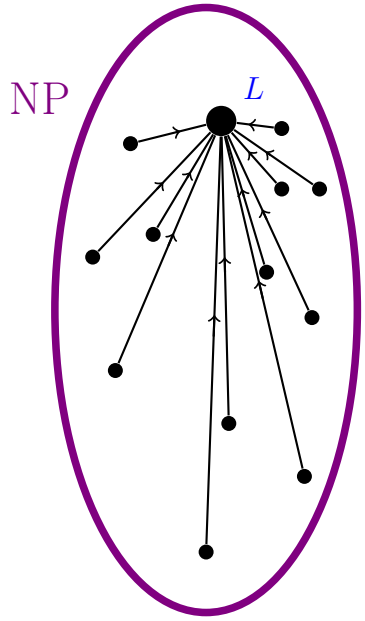
K can be transformed into any instances of L efficiently

** Every language in NP can transform or reduced to L in polynomial time

shows that L is at least as hard as any problems in NP

We illustrate this definition below. The first Venn diagram shows the class NP with an NP-complete language L shown as a dot within it. This language L is NP-complete because it is in NP and every other language in NP is polynomial-time reducible to it. We have shown just a few of those other languages, as the other dots in the diagram. Each one of these other languages has an arrow going to L, representing some polynomial-time reduction from that language to L. We will often picture NP-complete languages near the "top" of our depictions of NP, because they are in some sense among the hardest languages in NP. But it's a Venn diagram, not a geometric diagram; the shape of the ovals representing the language classes is irrelevant and the physical distances between languages on the page have no meaning.

** a language L is NP-complete if it's in NP and every problem in NP can be reduced to L in polynomial time



L is **NP-complete** because ...

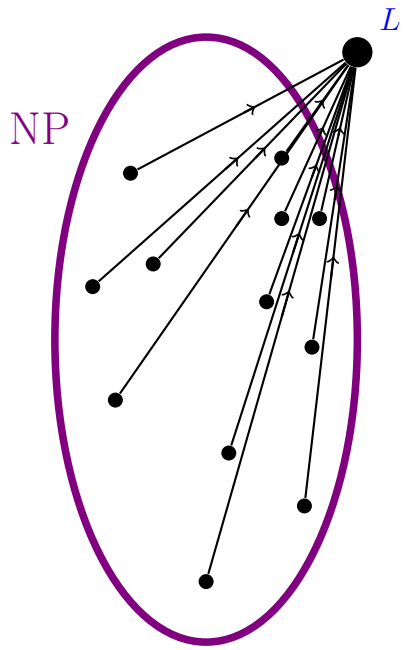
L is in NP,

and

everything in NP
is polynomial-time reducible to L

If a language is *not* NP-complete, then this could be for one (or both) of two reasons.

The first reason is that it might not be in NP: see the next diagram. That's enough to miss out on being NP-complete, even if every language in NP is polynomial-time reducible to it.

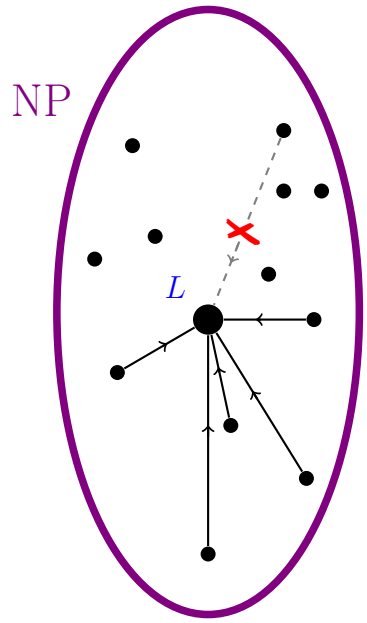


L is not NP-complete because ...

L is not in NP.

It doesn't matter if
everything in NP
is polynomial-time reducible to L

The second reason is that *not* all languages in NP might be polynomial-time reducible to L. There might be *some* languages in NP that are *not* polynomial-time reducible to L. See the next diagram.



L is not NP-complete because . . .

not everything in NP
is polynomial-time reducible to L

27.2 Properties of NP-completeness

Much of the importance of NP-completeness comes from the following theorem. This explains the central role that NP-complete languages play in the P-versus-NP question — both in efforts to resolve the question, and in motivating work on the question.

Theorem 69
Let L be any NP-complete language. There is a polynomial-time decider for L if and only if P = NP.

But whether P = NP or P ≠ NP, it is unknown / unconfirmed

Proof.
($\Longleftarrow$)
If P = NP, then every language in NP has a polynomial-time decider.
Since L is NP-complete, it must be in NP (using the first part of the definition of NP-completeness).
Therefore it's also in P.
Therefore (by definition of P), L has a polynomial-time decider.

($\Longrightarrow$)
Suppose L has a polynomial-time decider.
We know P $\subseteq$ NP.
It remains to show that, under our assumptions, NP $\subseteq$ P. Then we'll know that P = NP.

Let K be any language in NP. We will show that (under our assumptions) is also in P.
Since L is NP-complete, any language in NP is polynomial-time reducible to L.

Therefore $K \leq_P L$.

But we know that, if $K \leq_P L$ and L is in P, then K is in P too. (See Theorem 66 in Lecture 26.)

So K is in P.

We have shown that NP $\subseteq$ P, which completes the proof. □

Here are two more fundamental properties of NP. Proving them is good practice in using the properties of NP and polynomial-time reductions and in writing proofs.

Theorem 70

Let L be any NP-complete language. For every language K, K is in NP if and only if $K \leq_P L$.

one-way mapping reduction

Theorem 71

Let L be any NP-complete language. For every language K, K is in NP-complete if and only if $K \leq_P L$ and $L \leq_P K$.

two-way mapping reduction

27.3 A first NP-complete language

We have now spent some time discussing the definition of NP-complete languages and the fundamental properties of NP-completeness. We have mentioned the central role they play in the P-versus-NP question. But we have not yet shown that any NP-complete languages exist! Merely writing down a definition of NP-completeness, and noting how important such languages must be, is not enough to make them exist! So we must fill this gap.

Our first NP-complete language is

SATISFIABILITY := { satisfiable Boolean expressions in CNF }

The **Cook-Levin Theorem** states that SATISFIABILITY is NP-complete. This was proved independently in the following papers:

- Stephen Cook, The complexity of theorem-proving procedures, *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, Association for Computing Machinery, New York, 1971, pp. 151–158.
- Leonid A. Levin, Universal search problems (in Russian), *Problemy Peredachi Informatsii* **9** (no. 3) (1973) 115–116. Universal sequential search problems, *PINFTRANS: Problems of Information Transmission* 9, 1973.

To prove the Cook-Levin Theorem, we must show:

(a) SATISFIABILITY is in NP

This is the easy part. We need to give a polynomial-time verifier for SATISFIABILITY. In outline, this would work as follows.

- Input: a Boolean expression φ in CNF.
- Certificate: a truth assignnment to the variables of φ.
- Verification: check that each clause is satisfied by this truth assignment.

Then you just have to prove that the verification works and takes polynomial time.

(b) For every L in NP, $L \leq_P$ SATISFIABILITY.

This is much harder, or at least more time consuming. But you have already met all the main ideas in the proof.

In Lecture 28, we will see how to reduce from some specific NP-complete languages to SATISFIABILITY. This will build intuition as to how to reduce to SATISFIABILITY in general, which is good preparation for proving the Cook-Levin Theorem. Reducing to SATISFIABILITY is also a practical skill that can be used in real-world computational problem-solving.

In Lecture 29, we will prove the Cook-Levin Theorem.

Revision

Things to think about:

- If P = NP, which languages would be NP-complete?

Reading: Sipser, section 7.4, pp. 299–304.

Theorem 69

this is a condition ↑

Let L be any NP-complete language. There is a polynomial-time decider for L if and only if P = NP.

Theorem 69 have a statement that assumes P = NP as a condition. Let me clarify the implications of this theorem and why it's significant.

The theorem states: "Let L be any NP-complete language. There is a polynomial-time decider for L if and only if P = NP."

- If **P = NP is true**, it means that there exists a polynomial-time algorithm to solve all problems in NP, including NP-complete problems. This would indeed imply that there is a polynomial-time decider for any NP-complete language.

- If **P = NP is false**, it means that no polynomial-time algorithm can solve all problems in NP. In this case, Theorem 69 doesn't necessarily mean that there is a polynomial-time decider for any NP-complete language. Instead, it highlights a significant result in computational complexity theory: if P ≠ NP, it implies that NP-complete problems are inherently hard to solve in polynomial time, and no polynomial-time algorithm exists for them.

So, Theorem 69 is a **conditional statement** that highlights the profound implications of P vs. NP. It doesn't confirm the truth or falsehood of P = NP but rather demonstrates the consequences of each possibility:

- If P = NP, it implies that NP-complete problems are efficiently solvable.

- If P ≠ NP, it implies that NP-complete problems are inherently hard, and no efficient algorithm exists for them.

The theorem is not saying that we have confirmed that P = NP or that a polynomial-time decider for an NP-complete language exists; it's conditional. It suggests that if you were to discover such a polynomial-time decider for any NP-complete language, it would have profound implications for the field of computer science and complexity theory. In essence, it would prove that P = NP.

The unresolved question of whether P = NP or P ≠ NP is indeed one of the most significant unsolved problems in computer science, and it carries substantial implications for the complexity of various computational problems.

If P = NP, which languages would be NP-complete?

If P were equal to NP, it would imply that every problem in NP can be solved in polynomial time. In such a scenario, the concept of NP-completeness, as we understand it today, would lose its significance because every problem in NP would also belong to P. NP-complete problems are those that are at least as hard as the hardest problems in NP. However, if P were equal to NP, then every problem in NP would be equally easy to solve, and the distinction between NP and NP-complete would disappear.

In other words, if P = NP, there would be no NP-complete problems in the traditional sense because all problems in NP would be solvable in polynomial time, and there would be no need to distinguish between them in terms of complexity.

It's important to note that if P were proven to be equal to NP, it would be a groundbreaking result with significant implications for the field of computer science and cryptography, as many cryptographic systems rely on the assumption that P is not equal to NP. However, as of my last knowledge update in September 2021, this problem remains unresolved, and it's one of the most important open questions in the field of theoretical computer science.

Cook-Levin Theorem

The Cook-Levin Theorem, also known as the Cook-Levin Theorem or Cook's Theorem, is a fundamental result in computational complexity theory that established the concept of NP-completeness. This theorem was independently proved by Stephen Cook and Leonid Levin in the early 1970s.

Theorem (Cook-Levin Theorem):

> The Cook-Levin Theorem states that a problem is NP-complete if it is in NP and any problem in NP can be polynomial-time reduced to it.

A decision problem, such as the Boolean satisfiability problem (SAT), is NP-complete if and only if it is both in NP (i.e., solutions can be verified in polynomial time) and any problem in NP can be reduced to it in polynomial time.

Let's break down this theorem into its key components and provide a more detailed explanation:

1. **NP-Completeness**: A decision problem is NP-complete if it is one of the hardest problems in the complexity class NP. This means that if you can efficiently solve any NP-complete problem, you can efficiently solve all problems in NP. In other words, NP-complete problems are the "hardest" problems in NP.

2. **In NP**: For a problem to be NP-complete, it must first be in NP. This means that for any given proposed solution, there is a deterministic Turing machine that can verify the correctness of the solution in polynomial time. In the case of SAT, a proposed assignment of truth values to variables can be efficiently checked for correctness.

3. **Polynomial-Time Reduction**: The other part of the Cook-Levin Theorem states that any problem in NP can be reduced to the NP-complete problem in polynomial time. This means that there is a polynomial-time algorithm that can transform instances of any NP problem into instances of the NP-complete problem.

The importance of the Cook-Levin Theorem lies in the concept of NP-completeness. If you can demonstrate that a new problem is NP-complete (by showing it satisfies both parts of the theorem), it immediately implies that this problem is as hard as the hardest problems in NP. It's a significant concept because it allows us to classify problems in terms of their computational complexity and provides a basis for understanding which problems are likely to be inherently difficult to solve.

The Cook-Levin Theorem was a pivotal development in computational complexity theory and is the cornerstone of many practical and theoretical applications, including the development of algorithms, cryptography, and the study of the P vs. NP problem.

Lecture 28

NP-completeness: reductions to SATISFIABILITY

The Cook-Levin Theorem states that *any* language in NP can be polynomial-time reduced to SAT. To understand how this can be, we will look at some specific languages.

We begin with some general advice on how to do such reductions. Then we illustrate this approach with an extended example. We conclude with an indication of the overall strategy used in the Cook-Levin Theorem.

28.1 How to reduce to SAT

Suppose we have a language which we want to reduce to SAT.

Our overall approach to designing a reduction to SAT is as follows.

1. Introduce Boolean variables to describe the parts of the certificate.
2. (Possibly introduce other variables to help describe the conditions under which a certificate is valid.)
3. Translate the **rules of the language** into CNF.
4. Put it all together as an algorithm.

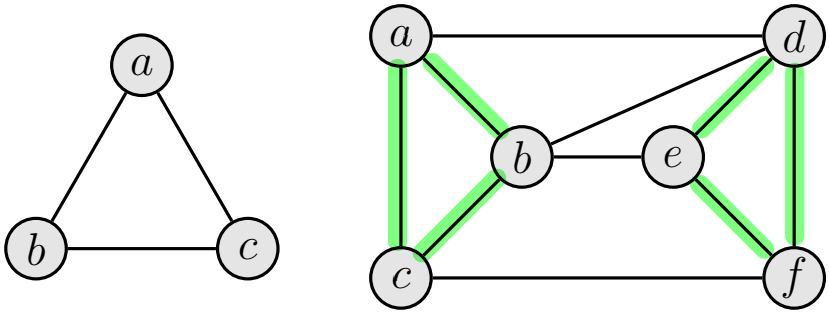
28.2 Extended example

To illustrate this approach, we will show that

$$\text{PARTITION INTO TRIANGLES} \leq_P \text{SATISFIABILITY},$$

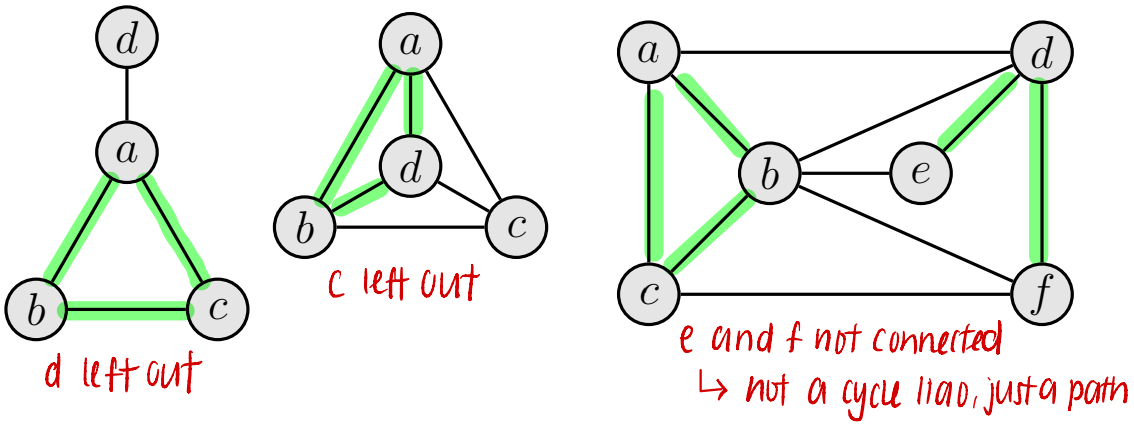
where PARTITION INTO TRIANGLES is the set of graphs G such that the vertex set of G can be partitioned into 3-sets (i.e., sets of size 3) such that each of these 3-sets induces a triangle in G. (In graph theory, a **triangle** is just a cycle consisting of three vertices and three edges.) In other words, for G to belong to this language, there must be a collection of triangles in G such that every vertex of G belongs to *exactly* one vertex of G.

Here are some examples of members of PARTITION INTO TRIANGLES.



The graph on the left is just a triangle, so a partition into triangles for it consists of just the single 3-set $\{a, b, c\}$. For the graph on the right, a partition into triangles consists of the two 3-sets $\{a, b, c\}$, $\{d, e, f\}$. Note that not every triangle needs to belong to such a partition; in this graph, the triangle induced by the 3-set $\{b, d, e\}$ is not used in our partition, and that's ok. The partition only needs *just enough* triangles to meet each vertex exactly once.

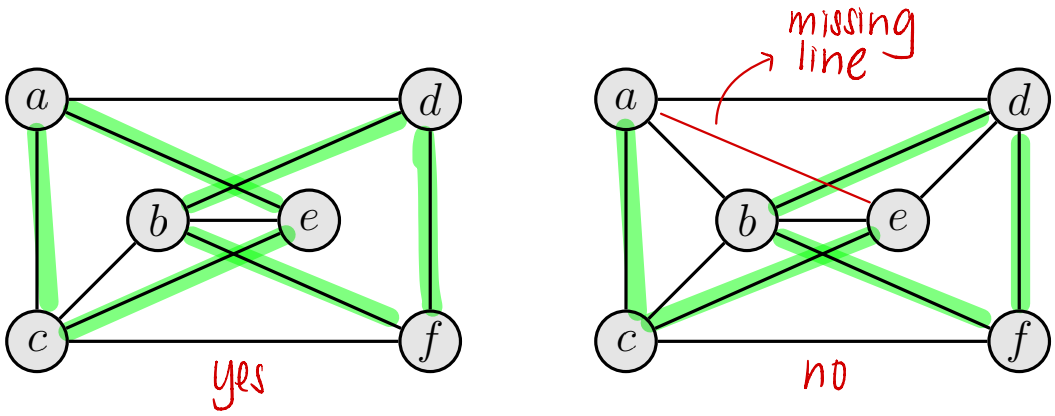
Here are some <u>non-members</u> of PARTITION INTO TRIANGLES.



The graph on the left cannot be in PARTITION INTO TRIANGLES because vertex d does not belong to *any* triangle, and also because its number of vertices is not a multiple of 3. The middle graph cannot be in PARTITION INTO TRIANGLES because it will need at least two triangles in any partition (because it has more than three vertices), yet every pair of triangles in the graph has at least one (in fact, two) vertices in common. It, too, has a number of vertices that is not a multiple of 3. For the graph on the right, some more thought is needed. Its number of vertices is six, which is a multiple of 3, so we cannot rule it out based purely on its number of vertices. Every vertex belongs to at least one triangle in the graph, so we cannot rule it out in the same way as we ruled out the left graph (which had a vertex that was not in any triangle). We see that vertex e belongs to *exactly* one triangle in the graph, namely the one with vertices $\{b, d, e\}$, so that 3-set *must* be one of the 3-sets

in any partition into triangles of this graph. But that leaves the vertices a, c, f, and they do not form a triangle. Therefore this graph has no partition into triangles.

For each of the following graphs, determine whether or not they belong to PARTITION INTO TRIANGLES.



One of them belongs to PARTITION INTO TRIANGLES, and once you have a partition into triangles for it, you can easily check it. This is because the language is in NP, and the partition into triangles constitutes a certificate that can be verified efficiently.

The other graph does not belong to PARTITION INTO TRIANGLES. To show that takes more effort. You will probably find yourself doing some step-by-step reasoning based on possible choices of which triangles belong to the partition. This is typical for non-members of languages in NP that are not known to belong to P.

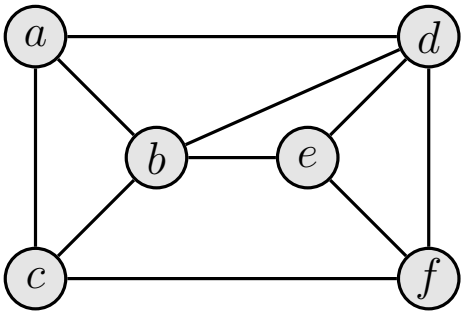
In our approach to designing a polynomial-time reduction to SAT, we first identify the type of certificate we are going to use and then define some Boolean variables to enable representation of any possible certificate.

In this case, the certificate is a specification of which triangles, in the graph, are in the partition.

For each triangle T in G, we introduce a Boolean variable x_T with the following intended interpretation.

$$x_T = \begin{cases} \mathsf{True}, & \text{if } T \text{ is in the partition;} \\ \mathsf{False}, & \text{otherwise.} \end{cases}$$

Suppose G is the following graph.



Its triangles are abc, abd, bde, def. So we use the variables

$$x_{abc},\ x_{abd},\ x_{bde},\ x_{def}\ .$$

In general, the number of triangles in a graph is at most the number of triples of vertices. So, if the graph has n vertices, then the number of triangles is $\binom{n}{3} = O(n^3)$. So the number of variables x_T that we introduce is $O(n^3)$. So we can list all these variables in polynomial time.

We now come to the heart of the design process: encoding the rules of the language into Boolean expressions in CNF that use our variables.

For PARTITION INTO TRIANGLES, the requirement is that the set of triangles must form a partition of V(G). As we observed above, this means that every vertex belongs to *exactly* one of the triangles in the partition. We have seen before (Lecture 2) that, to say "exactly", we can express it as a conjunction of "at least" and "at most". (This is just the observation that $=$ means both $\leq$ and $\geq$.) So we can express our requirement as a conjunction of two rules:

- every vertex belongs to at least one triangle, *and*
- no vertex belongs to more than one triangle.

Let's look at each of these in turn.

"Every vertex belongs to at least one triangle."

We can refine this statement a bit further by looking into what happens at each vertex in more detail. We obtain:

For each vertex: at least one of the triangles at that vertex must be included in the partition.

The local condition at the vertex — "at least one of the triangles ... must be included ..." — can be expressed as a disjunction (as we might expect from "at least").

So, for each vertex, we can use a clause of the form

$$x_{T_1} \vee x_{T_2} \vee \ldots \vee x_{T_k}$$

where $T_1, T_2, \ldots, T_k$ are the triangles at the vertex.

We have one of these clauses for each vertex, so the number of these clauses is just n, the number of vertices of the graph).

In our example graph, consider vertex a. It belongs to the two triangles abc and abd. So we use the clause $x_{abc} \vee x_{abd}$. Doing this for each vertex gives the following clauses:

Vertex a:	$x_{abc} \vee x_{abd}$
Vertex b:	$x_{abc} \vee x_{abd} \vee x_{bde}$
Vertex c:	x_{abc}
Vertex d:	$x_{abd} \vee x_{bde} \vee x_{def}$
Vertex e:	$x_{bde} \vee x_{def}$
Vertex f:	x_{def}

"No vertex belongs to more than one triangle."

We can rewrite this statement as

For each pair of triangles that have a vertex in common,
at most one of these two triangles is in the partition.

In other words,

For each pair of triangles that have a vertex in common,
at least one of these two triangles is not in the partition.

We express this in terms of our variables.
For each pair of triangles T_i, T_j that have a vertex in common, use a clause

$$\neg x_{T_i} \vee \neg x_{T_j}$$

The number of these clauses is bounded above by the number of pairs of triangles in G. So we have

$$\#\text{ of these clauses} \leq \binom{\#\text{ triangles}}{2} \leq \binom{\binom{n}{3}}{2} = O(n^6).$$

This is not the tightest possible bound[1], but it is good enough for a polynomial upper bound.

In our example graph, we have the following triangle pairs with a vertex in common, each with its associated clause.

abc, abd meet at vertex a:	$\neg x_{abc} \vee \neg x_{abd}$
abc, bde meet at vertex b:	$\neg x_{abc} \vee \neg x_{bde}$
abd, bde meet at vertices b, d:	$\neg x_{abd} \vee \neg x_{bde}$
abd, def meet at vertex d:	$\neg x_{abd} \vee \neg x_{def}$
bde, def meet at vertices d, e:	$\neg x_{bde} \vee \neg x_{def}$

We now have clauses for:

- every vertex belongs to at least one triangle, *and*
- no vertex belongs to more than one triangle.

Then we just take the conjunction of all clauses.

This gives a Boolean formula φ which is satisfiable if and only if the original graph has a partition into triangles.

For our example graph, we obtain

$$\begin{aligned}\varphi \;=\; & (x_{abc} \vee x_{abd}) \wedge (x_{abc} \vee x_{abd} \vee x_{bde}) \wedge (x_{abc}) \wedge (x_{abd} \vee x_{bde} \vee x_{def}) \wedge (x_{bde} \vee x_{def}) \wedge (x_{def}) \\ & \wedge (\neg x_{abc} \vee \neg x_{abd}) \wedge (\neg x_{abc} \vee \neg x_{bde}) \wedge (\neg x_{abd} \vee \neg x_{bde}) \wedge (\neg x_{abd} \vee \neg x_{def}) \wedge (\neg x_{bde} \vee \neg x_{def}).\end{aligned}$$

[1] For a tighter bound, we can loop over vertices and, for each vertex, just look at pairs of triangles that share that vertex. The number of triangles that contain a given vertex is $\leq \binom{n}{2}$, which is tighter than the bound $\binom{n}{3}$, so the number of clauses is $\leq n \cdot \binom{\max\ \#\text{ triangles at a given vertex}}{2} \leq n \cdot \binom{\binom{n}{2}}{2} = O(n^5)$

To make a polynomial-time reduction from this construction, we have to specify the mapping from G to φ as an algorithm, and show that it runs in polynomial time.

Here is the reduction as an algorithm.

Input: Graph G

1. For each triangle T of G:

   Create new variable x_T.

2. For each vertex v of G:

   Let $T_1, T_2, \ldots, T_k$ be the triangles at v.

   Make the clause $x_{T_1} \vee x_{T_2} \vee \ldots \vee x_{T_k}$.

3. For each pair of triangles T_i, T_j which share a vertex:

   Make a clause $\neg x_{T_i} \vee \neg x_{T_j}$

4. φ := conjunction of all these clauses.

5. Output φ

What can we say about its time complexity?

The main factor is the number of pairs of triangles that share a vertex, which is $O(n^6)$.

For each such pair, a small amount of work needs doing, to set up the clauses we need. This amount of work is certainly polynomial time, in fact $O(n)$ or even better.

So the reduction runs in polynomial time. So it's a polynomial-time reduction!

This completes our proof that

$$\text{PARTITION INTO TRIANGLES} \leq_P \text{SATISFIABILITY}.$$

28.3 Other reductions to SATISFIABILITY

We have looked at *one* particular language in NP and proved that it is polynomial-time reducible to SATISFIABILITY. So we have started our journey towards the Cook-Levin Theorem, which states that *every* language in NP is polynomial-time reducible to SATISFIABILITY. So we need to look at some more languages!

Other languages can be polynomial-time reduced to SAT by a similar approach.

Some good exercises of this type include

- 3-COLOURABILITY

- CUBIC SUBGRAPH:
  the set of graphs with a subgraph consisting entirely of vertices of degree 3

- HAMILTONIAN CIRCUIT

Experience with reducing to SAT suggests that any language in NP can be polynomial-time reduced to SAT. (This is the Cook-Levin Theorem.)

We will prove this in the next lecture. → lecture 29

It is not possible to prove the Cook-Levin Theorem by hand-crafting specialised reductions tailored to each specific language in NP, because there are infinitely many such languages.

Instead, we need a generic way of reducing from any specific language in NP to SAT.

Our proof strategy will be as follows.

Given L in NP, let M be a polynomial-time verifier for L.

Create variables for the certificate (considered as a binary string), and more variables to describe all the components of M at all possible time-steps.

Make clauses to capture the working of the verifier. These express the allowed configurations of the machine (i.e., the states, tape contents, and tape head positions), together with the allowed transitions from one time to the next, and forbid any others.

Show that the construction is indeed the desired polynomial-time reduction.

28.4 SAT Solvers

We conclude this lecture by mentioning a positive application of doing polynomial-time reductions to SATISFIABILITY to help solve practical problems.

A **SAT solver** is a program for solving SATISFIABILITY. To be useful in practice, these programs will do more than just decide if a given Boolean expression is satisfiable or not. In cases where they determine that the expression is satisfiable, they also return a satisfying truth assignment for the expression. if satisfiable = return satisfying truth assignment

SAT solvers do not solve SAT in polynomial-time. (If one did that, then we would know that P = NP.) They run in (worst case) exponential time, but they are carefully designed to exploit features of the kinds of SAT problems that people tend to need to solve in practice. So they can excel on some special cases of SAT that have been extensively studied and for which they have been tailored. There are always *some* cases of SAT for which they would take far too long, but they have developed to the point where SAT problems arising in practical applications are often found to be effectively solvable by SAT solvers.

If you have a polynomial-time reduction from your favourite NP problem to SATISFIABILITY then, in principle, you can use it, together with a SAT solver, to solve instances of your favourite problem. To do this, an instance of your problem is mapped, by the reduction, to an instance of SATISFIABILITY which is then given to the SAT solver. The SAT solution obtained, including a satisfying truth assignment, can be translated back to your problem to give a solution to it. To do this effectively in practice would require designing the polynomial-time reduction so that it is as efficient as possible, and in particular so that the Boolean expressions it produces are not unduly large.

Donald Knuth devotes a few hundred pages to SAT solving in *The Art of Computer Programming: Volume 4B: Combinatorial Algorithm, Part 2* (Addison-Wesley, Boston, Ma., 2023), §7.2.2.2., pp. 185 ff.

SageMath (`https://doc.sagemath.org`) includes a basic SAT solver in the function `is_satisfiable()`, which only reports whether a given Boolean expression is True or False, and (typically through installation of appropriate packages) some more powerful SAT solvers that also return a satisfying truth assignment.

Another SAT solver is PySAT (`https://pysathq.github.io`). A series of annual competitions among SAT solvers is reported at `http://www.satcompetition.org`.

Revision

Things to think about:

- Try doing one of the other polynomial-time reductions we've mentioned, e.g.,

$$\text{CUBIC SUBGRAPH} \leq_P \text{SATISFIABILITY}$$

- Try doing a polynomial-time reduction from this problem to SATISFIABILITY:

  $$\text{FA-Nonempty} := \{ A : A \text{ is a Finite Automation that accepts at least one string} \}.$$

  Try doing it in such a way that the logical expression you construct models the execution of the FA.
  Use Boolean variables to represent the letters in each position of the input string ...

Reading: Sipser, section 7.4, pp. 299–304.

Try doing one of the other polynomial-time reductions we've mentioned, e.g.,

$$\text{CUBIC SUBGRAPH} \leq_P \text{SATISFIABILITY}$$

Perform a polynomial-time reduction from the **Cubic Subgraph** problem to the **Satisfiability (SAT)** problem.

The **Cubic Subgraph** problem is about finding a subgraph of a graph G that is a cubic graph (every vertex has a degree of 3).

To do this reduction, we'll construct a SAT instance such that it is satisfiable if and only if the original graph G contains a cubic subgraph.

1. **Input**: We start with a graph G as the input for the **Cubic Subgraph** problem.

2. **Construction**: We'll construct a Boolean formula in SAT that models the condition of finding a cubic subgraph within G.

3. **Boolean Variables**: For each vertex in G, we create a set of Boolean variables to represent the presence of edges to its neighbors. For each vertex V in G and each pair of its neighbors U and W, we introduce a variable X_VUW, which is true if there's an edge between V and both U and W (i.e., V is connected to a cubic subgraph through U and W).

4. **Constraints**:

- For each vertex V in G, we add constraints to ensure that it is connected to exactly three other vertices in the subgraph. This is expressed as a SAT clause using the variables X_VUW for all possible pairs of neighbors U and W of V.

- We also include clauses to ensure that if a vertex V is in the cubic subgraph, it must have exactly three neighbors (each pair of neighbors implies the presence of an edge).

5. **Satisfiability Check**: We convert the logical formula into CNF format and check if the formula is satisfiable.

If the SAT instance is satisfiable, it means that there exists a cubic subgraph within the original graph G. The assignment of Boolean values to the variables will indicate which vertices and edges belong to this cubic subgraph.

This reduction is polynomial-time because the number of Boolean variables and constraints is linear in the size of the input graph G, making it a polynomial-time reduction from **Cubic Subgraph** to **SAT**.

Therefore, if you can efficiently solve the **SAT** problem, you can also solve the **Cubic Subgraph** problem, and this demonstrates that the **Cubic Subgraph** problem is NP-hard.

Try doing a polynomial-time reduction from this problem to SATISFIABILITY:

$$\text{FA-Nonempty} := \{ A : A \text{ is a Finite Automation that accepts at least one string} \}.$$

Try doing it in such a way that the logical expression you construct models the execution of the FA.
Use Boolean variables to represent the letters in each position of the input string ...

To perform a polynomial-time reduction from the problem **FA-Nonempty** (determining if a given finite automaton accepts at least one string) to **Satisfiability (SAT),** we can construct a logical expression that models the execution of the finite automaton. We'll use Boolean variables to represent the letters in each position of the input string and ensure that a satisfying assignment corresponds to an accepting run of the automaton.

Let's say we have a finite automaton A with Q states, an alphabet Σ, and a transition function δ.

1. **Boolean Variables**: For each possible configuration of the automaton at each step (state and symbol in the tape), we introduce a Boolean variable. We'll use variables q_i for the state at time i and variables x_i for the symbol in the input tape at time i.

2. **Initial Configuration**: We set the initial configuration of the automaton as a clause in our SAT expression. This clause would involve setting the appropriate q_0 variable to the initial state of the automaton and the x_0 variable to the first symbol in the input.

3. **Transition Constraints**: For each time step, we add constraints in the SAT expression that enforce the transition rules of the automaton. These constraints ensure that if the automaton is in state q_i and reads symbol x_i at time step i, it should transition to state q_(i+1) and read symbol x_(i+1) at time step (i+1) as defined by δ.

4. **Accepting State**: We add a clause that specifies that the automaton must be in an accepting state at some time t. This clause ensures that the automaton accepts at least one string.

5. **Formula Composition**: We combine all the clauses and constraints into a single logical formula in conjunctive normal form (CNF), which is the standard format for SAT instances.

Now, if the logical formula is satisfiable, it means there exists an assignment of the Boolean variables that models an accepting run of the finite automaton. If the formula is unsatisfiable, it means that no such run exists, indicating that the automaton does not accept any string.

This reduction can be done in polynomial time because it involves a linear number of variables and constraints corresponding to the number of states and time steps in the automaton. If the constructed SAT formula is satisfiable, it demonstrates that the finite automaton has an accepting run, proving the non-emptiness of the language it recognizes.

Lecture 29

NP-completeness: the Cook-Levin Theorem

theorum is examinable but proof is not

In Lecture 25 (p. 270) we introduced the language

$$\text{SATISFIABILITY} := \{\text{satisfiable Boolean expressions in CNF}\}$$

and noted that it belongs to NP. Later in that lecture, it was identified as one of a number of languages in NP that are not known to belong to P.

In Lecture 27 (p. 298), we stated the Cook-Levin Theorem.

Theorem 72 (Cook-Levin Theorem: Cook, 1971; Levin, 1973)
SATISFIABILITY is NP-complete.

Stephen Cook (b. 1939) in 1968

Leonid Levin (b. 1948)

29.1 Starting the proof

We now prove that theorem. The proof follows the strategy outlined near the end of the previous lecture. The proof is long, with lots of detail. But, throughout, the ideas used are ideas we have used many times previously. The theme is to break the operation of a polynomial-time verifier down into a low-level description that can be expressed as a conjunction of clauses.

Proof.

We have already seen that SATISFIABILITY is in NP. It remains to show that every language in NP is polynomial-time reducible to SATISFIABILITY.

Let L be any language in NP.

We must give a polynomial-time reduction from L to SATISFIABILITY.

Let V be a Turing machine that is a polynomial-time verifier for L. Assume its input alphabet is {a,b}, its tape alphabet is {a,b,#,Δ}, its number of states is p, and its states are numbered $1, 2, \ldots, p$, with (as usual) state 1 being the Start State and state 2 being the Accept State.

Initially, the tape of V contains two strings, x and y:

- x is the input string whose membership, or not, of L is under consideration;
- y is the certificate.

Assume that x and y are separated on the tape by #. So the tape initially holds the string $x\#y$.

The fact that V is a **verifier** for L means:

V takes 2 strings (x,y) and accepts if and only if there exists a y that makes it accept the membership condition for L

$$x \in L \quad \text{if and only if} \quad \exists y : V(x, y) \text{ accepts.}$$

The fact that V runs in **polynomial time** means:

T(n) n = length of input x

$$\exists N, c, k \;\; \forall x \text{ such that } |x| \geq N \;\; \forall y \; : \;\; t_V(x, y) \;\leq\; c\,|x|^k.$$

For convenience, put $T(n) := \lfloor c\, n^k \rfloor$. This gives us an integer-valued polynomial upper bound for the time taken when $|x| = n$.

We will describe the *entire computation* of V starting with $x\#y$, by a Boolean expression φ_x in Conjunctive Normal Form.

We must ensure:

$$\exists y : V(x, y) \text{ accepts} \quad \text{if and only if} \quad \exists \text{ truth assignment} : \varphi_x \text{ is True.}$$

verifier acceptance ← $V(x, y)$ accepts

We will express the proposition

construction of φ_x: reduction aim to construct Boolean expression φ_x that captures the computation of the verifier V

as a conjunction of more specific propositions, and keep doing so, until all our propositions are so specific and detailed that we can express them in Boolean logic, as conjunctions of clauses in a CNF expression. That expression will be φ_x.

the reduction breaks down the computation of V to smaller, specific propositions to be easily expressed in Boolean logic

29.2 Boolean variables

combined into single Boolean expression in conjunctive normal form (CNF)

eg. state transitions, acceptance conditions

To begin with, we need Boolean variables that describe every possibility for every little piece of V at every possible time during the computation.

The next table lists our variables and their intended meaning. If we were to denote the variables by letters, we might represent them as in the first column. But, for the rest

main goal: make sure there exists a y that makes verifier accepts if and only if there exist a truth assignment that makes φ_x = TRUE

of the proof, we will denote the variables by small pictorial symbols that will, hopefully, indicate their purpose visually. The variables representing states are represented by a small circle, because we use small circles to represent states in diagrams of Turing machines and other automata. The variables representing tape cells are represented by small squares, since we depict tape cells as squares in diagrams of Turing machine tapes. The variables representing the tape head are represented by a small pentagon shaped like the way we have been drawing tape heads in diagrams of Turing machines. So, each variable is represented by a small picture of the thing it represents, together with appropriate subscripts.

Note that, when we state an *intended meaning* for a variable, *stating* this is *not sufficient* to ensure that the variable *actually has* that meaning! Merely asserting something does not make it true. Stating the intended meaning is like a comment statement in a program: it hopefully makes the ideas in the proof easier to follow, but it makes no contribution to the logic of the proof.

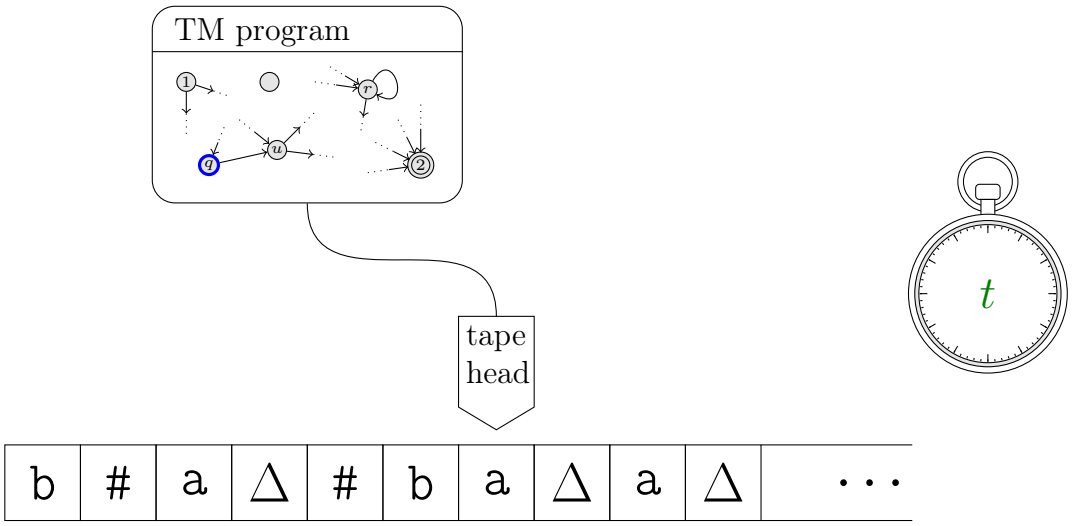
variable	variable	intended meaning	
$Q_{t,q}$	$\bigcirc_{t,q}$	At time t, the machine is in state q.	$1 \le t \le T(n), \quad 1 \le q \le p.$
$S_{t,s,\ell}$	$\square_{t,s,\ell}$	At time t, tape cell s contains letter ℓ.	$1 \le t \le T(n), \quad 1 \le s \le T(n),$ $\ell \in \{\mathtt{a}, \mathtt{b}, \Delta, \#\}$
$H_{t,s}$	$\bigtriangledown_{t,s}$	At time t, Tape Head is scanning tape cell s.	$1 \le t \le T(n), \quad 1 \le s \le T(n).$

At this point, you should count the variables we are introducing, and check that the number of them is polynomially bounded, in n.

Here is a picture of a Turing machine, to help visualise what we want the variables to describe. Note the usual Turing machine components: the program at top left, which we have often represented as a state diagram; the tape head, through which the program interacts with the tape; and the tape, with each cell containing a letter (or a blank, denoted by Δ, if it is empty). Note also the clock on the right, showing the current time, which here is t.

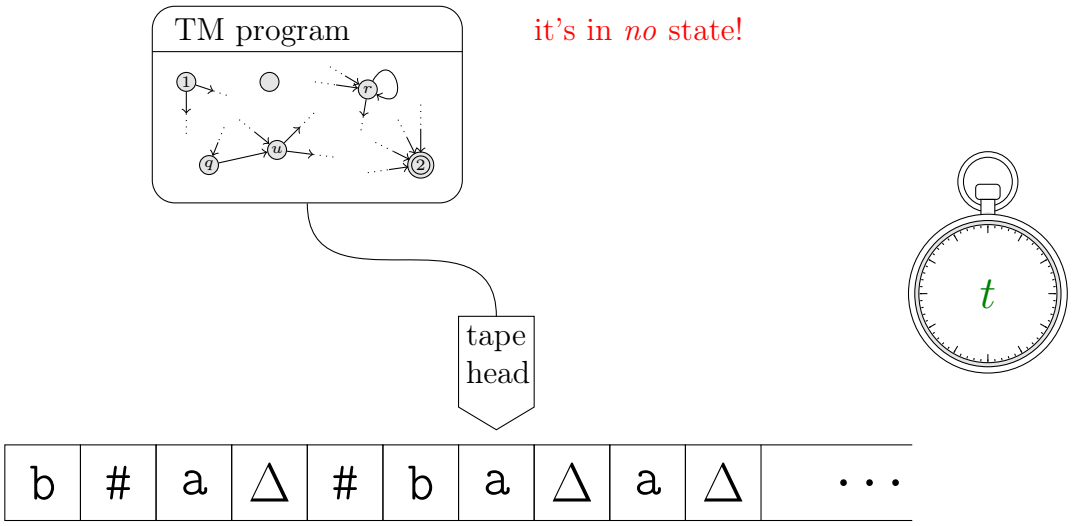
This diagram depicts a Turing machine's configuration at some time t. At that time, this particular Turing machine is in state q (highlighted in blue in the TM program), with its tape head scanning the seventh tape cell (which happens to contain a), and the tape contents being b#aΔ#baΔaΔ$\cdots$. The Turing machine must be in *exactly* one state, it must be scanning *exactly* one tape cell, and each tape cell must contain *exactly* one letter. (The changes in the Turing machine configuration, from one time to the next, must correctly follow an appropriate transition, too. We will deal with that later in the proof.)

b and a separated on the tape by * ⟶ b*a

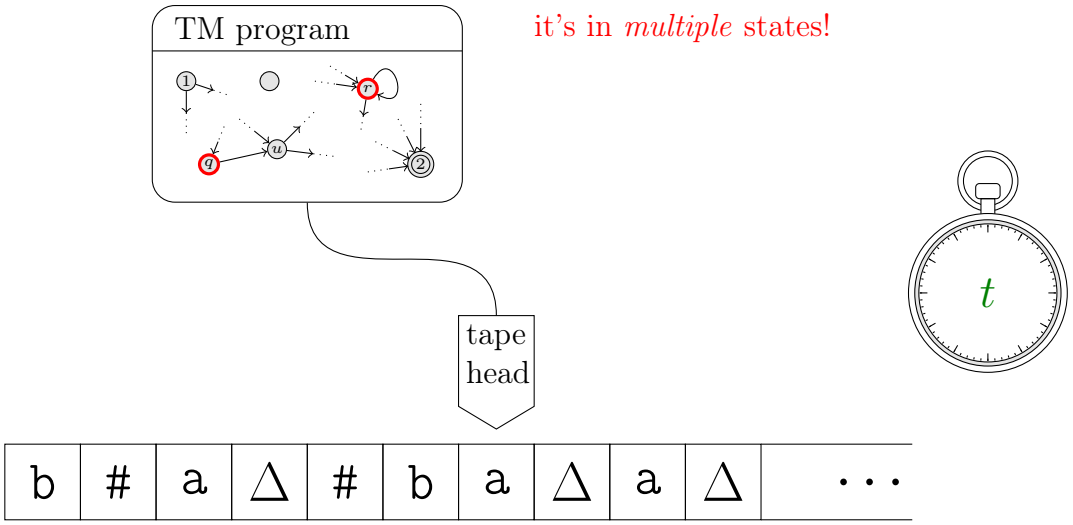


However, if we just let the variables loose, so to speak — that is, if we don't add any Boolean expressions to ensure that the variables act like a proper Turing machine — then we might get all sorts of chaos!

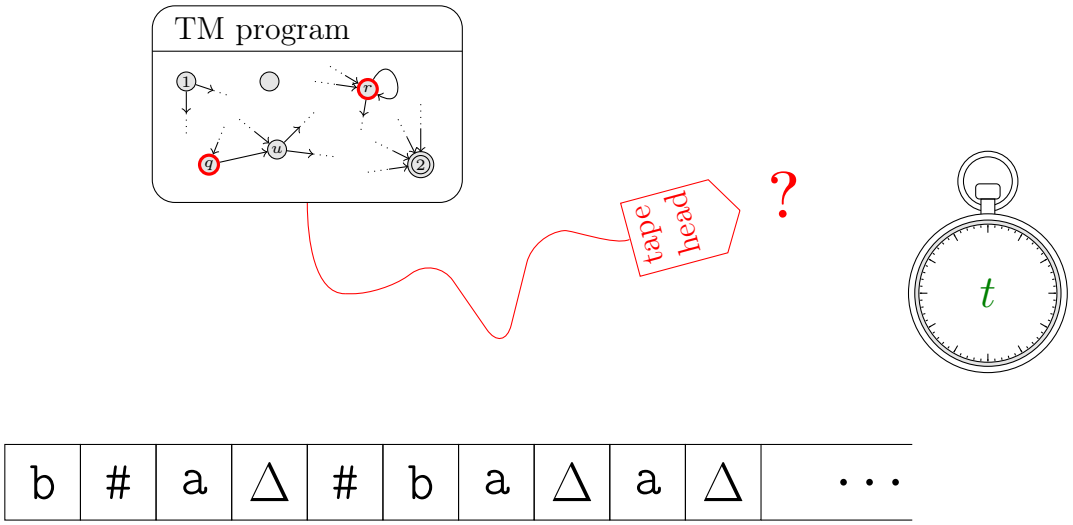
It's possible that the "machine" described by the variables might be in no state at all! The following diagram illustrates this possibility, with no state in the Turing machine program being highlighted. In terms of our variables, this would happen when, at time t, the variable $\bigcirc_{t,q} = \mathsf{False}$ for all $q = 1, 2, \ldots, p$.



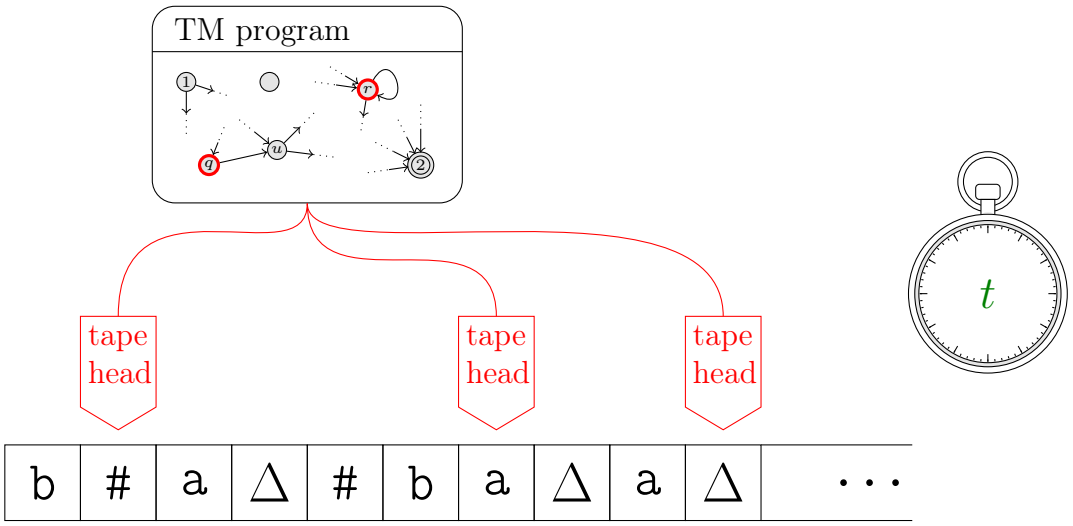
It's also possible that the "machine" is in more than one state at the same time! To illustrate this, we highlight both states q and r in the next diagram, but do so in red to indicate that this should not be allowed to happen. In terms of our variables, this situation arises when, at time t, the variable $\bigcirc_{t,q} = \mathsf{True}$ for two or more values of q.



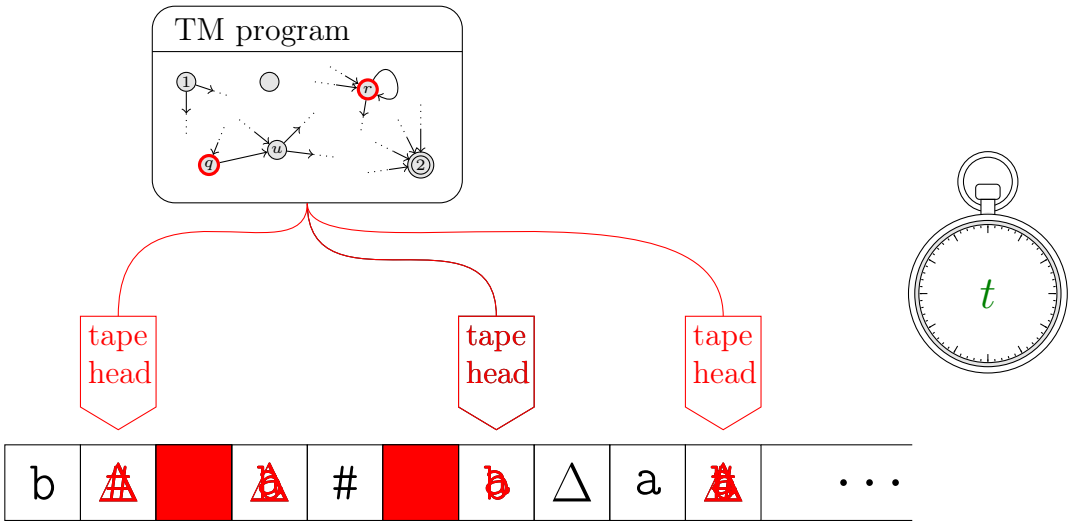
The tape head might not be scanning any tape cell at all! In terms of our variables, this would happen when, at time t, the variable $\heartsuit_{t,s} = \mathsf{False}$ for all s.



Or the tape head might be scanning two or more cells at once! In terms of our variables, this would happen when, at time t, the variable $\heartsuit_{t,s} = \mathsf{True}$ for two or more values of s. In the next diagram, we have $\heartsuit_{t,2} = \heartsuit_{t,7} = \heartsuit_{t,10} = \mathsf{True}$.



Some tape cells might contain *nothing at all* (not even a blank!). Others might contain *too much*, i.e., two or more letters (including possibly Δ) simultaneously. In terms of our variables, this would happen when, at time t and tape cell s, the variable $\square_{t,s,\ell} = \mathsf{False}$ for all ℓ (cell s has *nothing* at time t), or $\square_{t,s,\ell} = \mathsf{True}$ for two or more values of ℓ (cell s has *two or more things in it* at time t). In the next diagram, tape cell 2 contains both # and Δ, which corresponds to $\square_{t,s,\#} = \square_{t,s,\Delta} = \mathsf{True}$, while tape cell 3 has *nothing at all*, not even a blank, which corresponds to $\square_{t,s,\mathtt{a}} = \square_{t,s,\mathtt{b}} = \square_{t,s,\#} = \square_{t,s,\Delta} = \mathsf{False}$.



To prevent the sort of chaos we have just described, we need to write some clauses which, when combined using conjunction, ensure that the Turing machine we are describing operates correctly. It is convenient to consider two types of conditions that we want to express in logic:

- **static conditions** which ensure that, at every time, the Turing machine configuration is a valid Turing machine configuration (without worrying about how the configurations at one time relate to configurations at the next time);
  → maintain consistency & integrity of computation
- **dynamic conditions** which ensure that the *changes* in the Turing machine configuration, from one time to the next, always happen in accordance with the Turing machine's program.

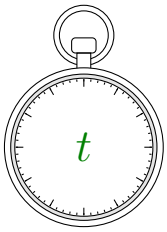
29.3 Static conditions

Static conditions include some general conditions that apply at all times and some extra conditions at the start and end of the computation. These conditions may be summarised as follows.

- For every time t: **The TM configuration is sane.**
- At time 0: **The initial set-up is correct.**
- At time $T(n)$: **The TM has accepted.**

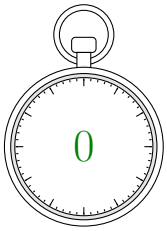
Each of these conditions may be broken down further as a conjunction of more specific conditions, each pertaining to state, or to tape head position, or to tape cell contents, as follows (with the time it pertains to shown in the clocks on the right).

For every time t: **The TM configuration is sane.**



- The machine is in exactly one state.
- The Tape Head is in exactly one position.
- For every tape cell s, the cell contains exactly one letter.

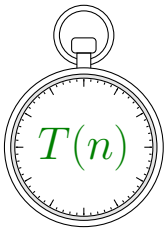
At time 0: **The initial set-up is correct.**



- The machine is in the Start state.
- The Tape Head is scanning the first tape cell.
- Tape cells 1 to n contain the letters of x, and tape cell $n+1$ contains #.

At time $T(n)$: **The TM has accepted.**

- The machine is in the Accept state.



This gives us seven conditions to capture in Boolean logic.
We consider each of these in turn.

For every time t, the machine is in exactly one state.

As usual, we express "exactly" as a conjunction of "at least" and "at most". This enables us to express the condition as a conjunction of clauses.

- For every time t, the machine is in at least one state.

$$\bigcirc_{t,1} \vee \bigcirc_{t,2} \vee \cdots \vee \bigcirc_{t,p}$$

- For every time t, the machine is in at most one state.

for each pair of states q, r, the machine is not in state q or it's not in state r.

$$(\neg\bigcirc_{t,q} \vee \neg\bigcirc_{t,r})$$

Joining all these clauses of size two together, for time t, we obtain the expression for the "at most" part:

$$\begin{aligned}(\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,2}) \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,3}) \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,p}) \\ \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,3}) \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,p}) \\ \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,p}) \\ \ddots \qquad\qquad \vdots \\ \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,p})\end{aligned}$$

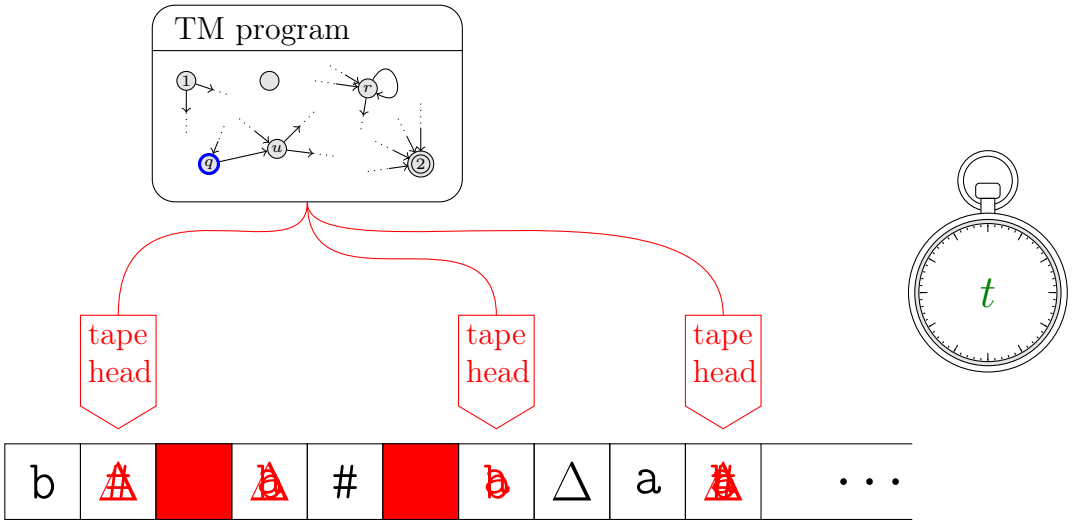
Now we join the "at least" clause with the long conjunction for "at most", giving the entire expression for the state at time t:

$$\begin{aligned}&(\bigcirc_{t,1} \vee \bigcirc_{t,2} \vee \cdots \vee \bigcirc_{t,p}) \\ &\quad \wedge \ (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,2}) \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,3}) \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,1} \vee \neg\bigcirc_{t,p}) \\ &\qquad \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,3}) \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,2} \vee \neg\bigcirc_{t,p}) \\ &\qquad \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,4}) \wedge \cdots \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,p}) \\ &\qquad\qquad \ddots \qquad\qquad \vdots \\ &\qquad\qquad \wedge (\neg\bigcirc_{t,3} \vee \neg\bigcirc_{t,p})\end{aligned}$$

Then form:

$$(\text{expression for } t=0) \wedge (\text{expression for } t=1) \wedge \cdots\cdots \wedge (\text{expression for } t=T(n))$$

This ensures that the Turing machine we are modelling in logic is always in exactly one state. But it might still have problems with its tape head or tape contents.



For every time t, the Tape Head is in exactly one position.

We express this condition as a conjunction of clauses as follows.

- For every time t, the Tape Head is in at least one position.

$$\heartsuit_{t,1} \vee \heartsuit_{t,2} \vee \cdots \vee \heartsuit_{t,T(n)}$$

- For every time t, the Tape Head is in at most one position.

for each pair of tape cells s_1, s_2, the Tape Head is not at cell s_1 or it's not at cell s_2

$$(\neg\heartsuit_{t,s_1} \vee \neg\heartsuit_{t,s_2})$$

Joining all these clauses of size two together, for time t, we obtain the expression for the "at most" part:

$$\begin{aligned}(\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,2}) \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,3}) \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,T(n)}) \\ \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,3}) \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,T(n)}) \\ \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,T(n)}) \\ \ddots \qquad\qquad \vdots \\ \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,T(n)})\end{aligned}$$

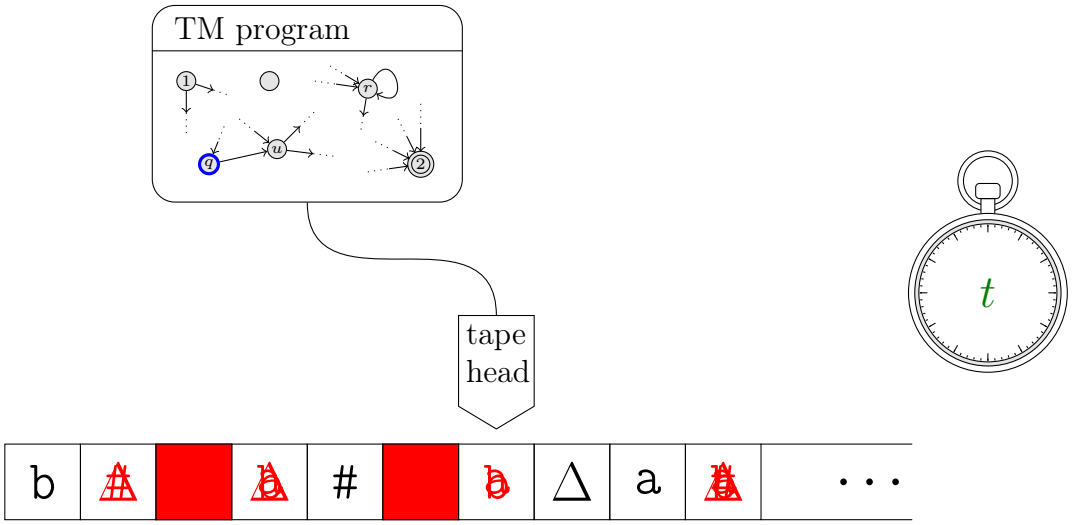
Now we join the "at least" clause with the long conjunction for "at most", giving the entire expression for tape head position at time t:

$$
\begin{aligned}
&(\heartsuit_{t,1} \vee \heartsuit_{t,2} \vee \cdots \vee \heartsuit_{t,T(n)}) \\
&\quad \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,2}) \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,3}) \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,1} \vee \neg\heartsuit_{t,T(n)}) \\
&\quad \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,3}) \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,2} \vee \neg\heartsuit_{t,T(n)}) \\
&\quad \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,4}) \wedge \cdots \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,T(n)}) \\
&\quad \ddots \qquad\qquad \vdots \\
&\quad \wedge (\neg\heartsuit_{t,3} \vee \neg\heartsuit_{t,T(n)})
\end{aligned}
$$

Then form:

$$(\text{expression for } t = 0) \wedge (\text{expression for } t = 1) \wedge \cdots\cdots \wedge (\text{expression for } t = T(n))$$

This ensures that the tape head of our Turing machine is always scanning exactly one tape cell. But it might still have problems with its tape contents.



For every time t and tape cell s, the cell contains exactly one letter.

We express this condition as a conjunction of clauses as follows, keeping in mind that we now have to vary s over all tape cell positions, as well as varying t over all possible times.

- For every time t and cell s, the cell contains at least one letter.

$$\square_{t,s,\mathsf{a}} \vee \square_{t,s,\mathsf{b}} \vee \square_{t,s,\#} \vee \square_{t,s,\Delta}$$

- For every time t and cell s, the cell contains at most one letter.

for each pair of letters ℓ, m, the cell doesn't contain ℓ or it doesn't contain m

$$(\neg\square_{t,s,\ell} \vee \neg\square_{t,s,m})$$

Joining these clauses of size two together, for time t and cell s, we obtain the expression for the "at most" part:

$$\begin{aligned}(\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\mathtt{b}}) \wedge (\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\#}) \wedge (\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\Delta}) \\ \wedge (\neg\square_{t,s,\mathtt{b}} \vee \neg\square_{t,s,\#}) \wedge (\neg\square_{t,s,\mathtt{b}} \vee \neg\square_{t,s,\Delta}) \\ \wedge (\neg\square_{t,s,\#} \vee \neg\square_{t,s,\Delta})\end{aligned}$$

Now we join the "at least" clause with the long conjunction for "at most", giving the entire expression for the contents of tape cell s at time t:

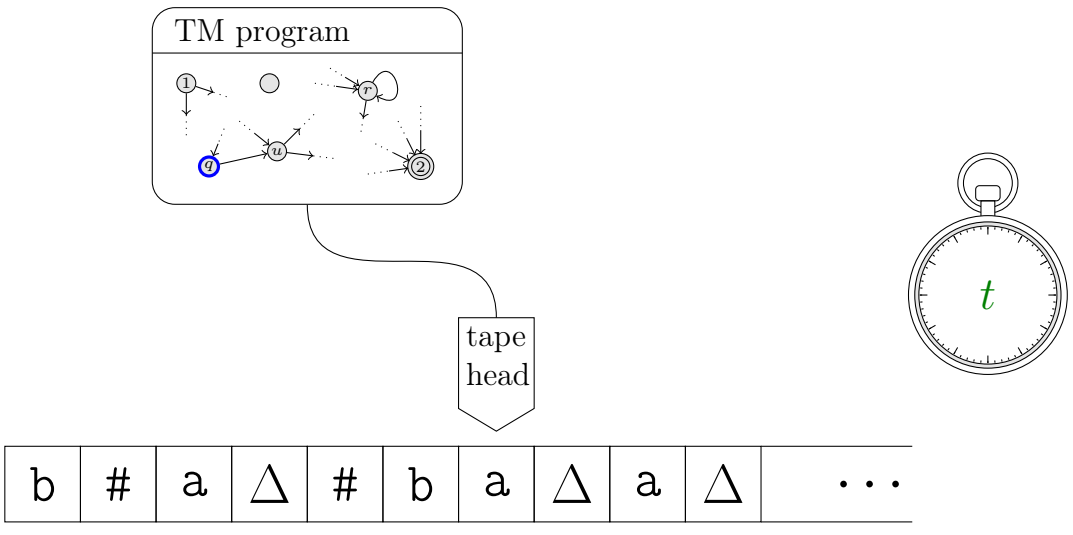
$$\begin{aligned}(\square_{t,s,\mathtt{a}} \vee \square_{t,s,\mathtt{b}} \vee \square_{t,s,\#} \vee \square_{t,s,\Delta}) \\ (\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\mathtt{b}}) \wedge (\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\#}) \wedge (\neg\square_{t,s,\mathtt{a}} \vee \neg\square_{t,s,\Delta}) \\ \wedge (\neg\square_{t,s,\mathtt{b}} \vee \neg\square_{t,s,\#}) \wedge (\neg\square_{t,s,\mathtt{b}} \vee \neg\square_{t,s,\Delta}) \\ \wedge (\neg\square_{t,s,\#} \vee \neg\square_{t,s,\Delta})\end{aligned}$$

Then form:

$$\begin{array}{ccccc}
(\text{expression for } t=0,\ s=0) & \wedge & \cdots\cdots & \wedge & (\text{expression for } t=0,\ s=T(n)) \quad \wedge \\
(\text{expression for } t=1,\ s=0) & \wedge & \cdots\cdots & \wedge & (\text{expression for } t=1,\ s=T(n)) \quad \wedge \\
\vdots & & \vdots & & \vdots \\
(\text{expression for } t=T(n),\ s=0) & \wedge & \cdots\cdots & \wedge & (\text{expression for } t=T(n),\ s=T(n))
\end{array}$$

This ensures that every tape cell of our Turing machine always contains exactly one symbol (including the possibility of Δ).

So we have brought order to chaos using Boolean logic in CNF, ensuring that, at every time t, the Turing machine configuration is sane, in that our logical expression represents a Turing machine configuration that is, in itself, valid.



We now attend to the three special conditions at time 0. These are much simpler. For each condition, we give a Boolean expression that represents it.

At time 0, the machine is in the Start state.

$$(\bigcirc_{0,1})$$

At time 0, the Tape Head is scanning the first tape cell.

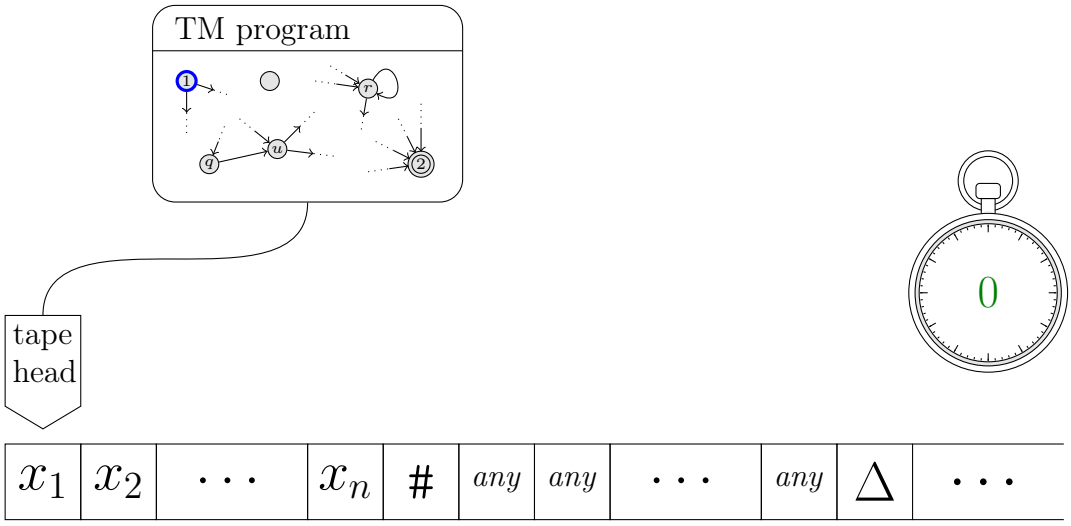
$$(\heartsuit_{0,1})$$

At time 0, tape cells 1 to n contain the letters of x, and cell $n+1$ contains #.
Suppose $x = x_1 x_2 \cdots x_n$, where each $x_i \in \{\mathtt{a}, \mathtt{b}\}$.

$$(\square_{0,1,x_1}) \wedge (\square_{0,2,x_2}) \wedge (\square_{0,3,x_3}) \wedge \cdots \wedge (\square_{0,n,x_n}) \ \wedge \ (\square_{0,n+1,\#})$$

We have now forced the Turing machine to start in state 1, with tape head at the first tape cell, and with the input string x on the input tape followed by the separator #, at time 0.

We do not need (or want) to stipulate any initial tape cell contents for the portion of the tape devoted to the certificate y. The construction of our big Boolean expression φ_x must depend *only* on x. Remember that polynomial-time reductions must not use the certificate (which they don't know about).

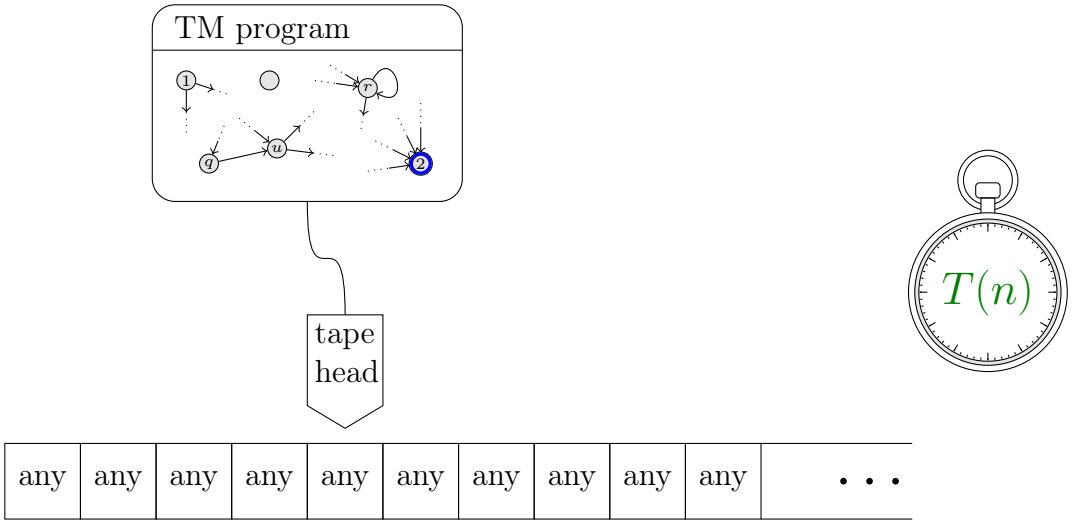


Our last static condition simply requires that the TM has accepted at the end of the computation. This is easily stated in logic.

At time $T(n)$, the machine is in the Accept state.

$$(\bigcirc_{T(n),2})$$

We make no specification as to where the tape head might be, or what the tape contents might be, at the end.

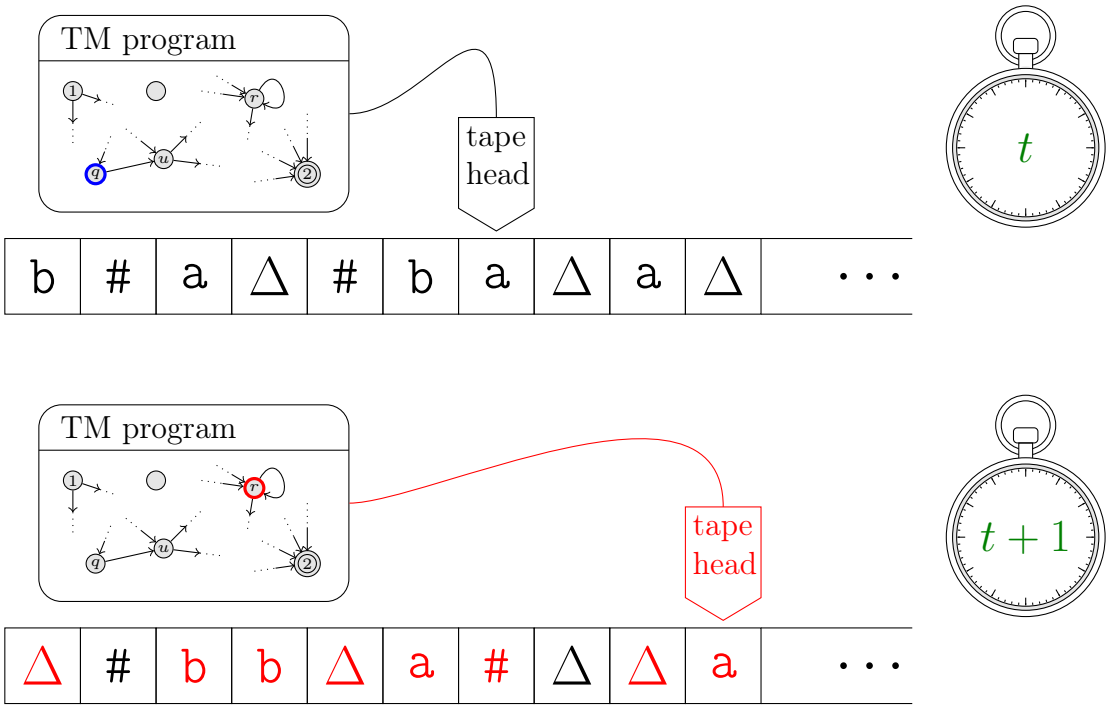


That concludes our representation of the static conditions as a Boolean expression in CNF.

ensure turing machine operates strictly in accordance to its program

29.4 Dynamic conditions

The expression we have constructed so far says nothing about what happens to the Turing machine configuration as we change from time t to time $t+1$. It is quite possible that the two configurations may bear no relation to each other. The state at time t (call it q) and the state at time $t+1$ (call it r) may have no transition going from the former to the latter (or, if there is a transition, it might not match the current letter being read). The tape head at time $t+1$ may be a long way away from where it is at time t, or it may be at the same tape cell (which is not allowed in our Turing machine model); even if it is at an adjacent tape cell, it might not be in the direction that corresponds to the transition that applies in the current situation (e.g., it might be to the right, when the transition applicable at time t stipulates a move to the left). The tape contents may be completely different, with tape cells changing their contents even if the tape head is not there, or the change at the tape head might be different to the one specified in whatever transition is applicable. We illustrate this "disconnect" between the two successive configurations in the next diagram, where things at time $t+1$ that are inconsistent with the configuration at time t are shown in red.



So we need more Boolean logic to specify how the TM changes from time t to $t+1$. The changes that can and cannot occur are captured by the following two conditions.

- **Cell content can only change at the tape head.**
- **Things change according to transitions.**

We look at each of these in turn.

29.4.1 Cell content can only change at the tape head.

This means that:

> For every time t and tape cell s:
> if the machine is not scanning tape cell s at time t,
> then the letter in this tape cell stays the same from time t to $t+1$.

This may be rewritten:

> For every time t and tape cell s:
> for each letter $\ell \in \{\mathtt{a}, \mathtt{b}, \#, \Delta\}$:
> if the machine is not scanning tape cell s at time t,
> and the letter in tape cell s at time t is ℓ,
> then the letter in this tape cell at time $t+1$ is also ℓ.

So, for each ℓ:

$$(\neg \boxed{\vee}_{t,s} \wedge \square_{t,s,\ell}) \Longrightarrow \square_{t+1,s,\ell}$$

In CNF, this is

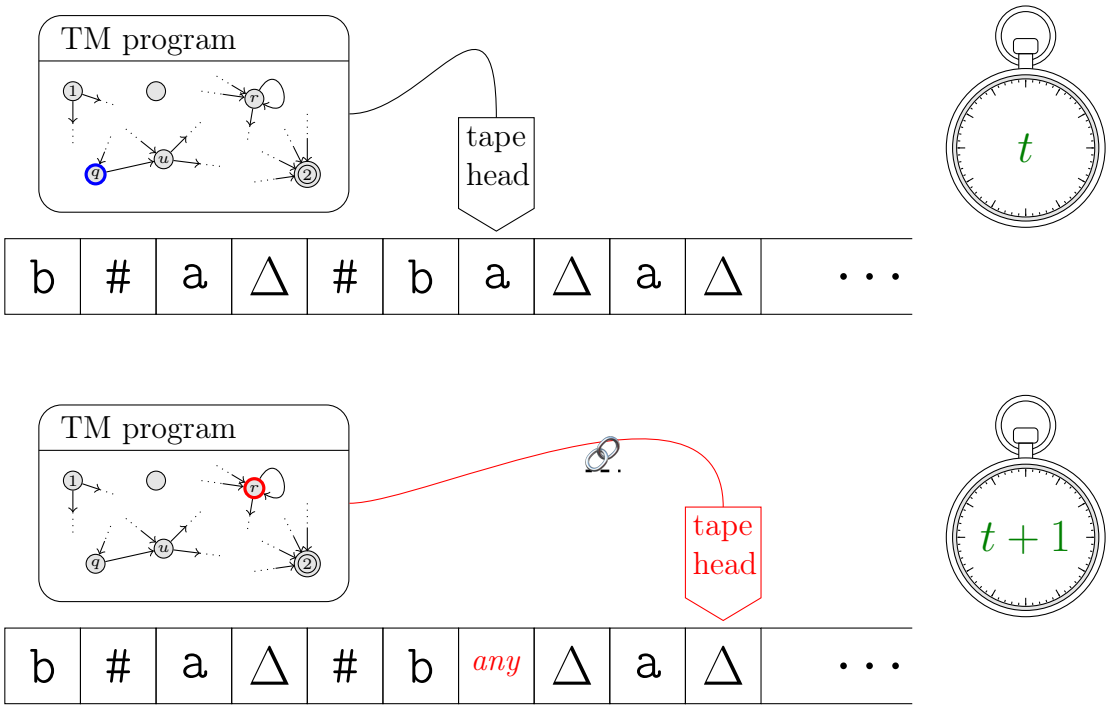
$$\boxed{\vee}_{t,s} \vee \neg \square_{t,s,\ell} \vee \square_{t+1,s,\ell}$$

Combining these, using conjunction, over all $\ell \in \{\mathtt{a}, \mathtt{b}, \#, \Delta\}$ gives:

$$\begin{array}{ll} & (\boxed{\vee}_{t,s} \vee \neg \square_{t,s,\mathtt{a}} \vee \square_{t+1,s,\mathtt{a}}) \\ \wedge & (\boxed{\vee}_{t,s} \vee \neg \square_{t,s,\mathtt{b}} \vee \square_{t+1,s,\mathtt{b}}) \\ \wedge & (\boxed{\vee}_{t,s} \vee \neg \square_{t,s,\Delta} \vee \square_{t+1,s,\Delta}) \\ \wedge & (\boxed{\vee}_{t,s} \vee \neg \square_{t,s,\#} \vee \square_{t+1,s,\#}) \end{array}$$

This brings most of the tape under control by forcing it not to change from time t to time $t+1$.

So, comparing our pictures of the TM configurations at times t and $t+1$, things are improving, though we still may have problems with the state, the tape head, and the contents of the tape cell being scanned by the tape head at time t.



We now look at the transitions.

29.4.2 Things change according to transitions.

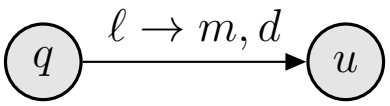
This means that:

For every time t, tape cell s, state q and letter ℓ,
if the machine is in state q, reading letter ℓ, and scanning tape cell s,
then at time $t+1$,

- the state and letter are as given by the transition for q and ℓ,
- the tape cell being scanned is $s-1$ or $s+1$ according to the direction (Left or Right) specified by that transition. Another way to say this is that the tape cell being scanned is $s+\sigma$ where

$$\sigma := \begin{cases} +1, & \text{if } d \text{ is Right;} \\ -1, & \text{if } d \text{ is Left.} \end{cases}$$

Suppose we have a transition from state q to state r labelled by $l \to m, d$, where ℓ, m are letters and d is a direction (Left or Right).



Then the effect of the transition is given by the following expressions.

$$\begin{aligned} (\bigcirc_{t,q} \wedge \square_{t,s,\ell} \wedge \heartsuit_{t,s}) &\implies \bigcirc_{t+1,u} \\ (\bigcirc_{t,q} \wedge \square_{t,s,\ell} \wedge \heartsuit_{t,s}) &\implies \square_{t+1,s,m} \\ (\bigcirc_{t,q} \wedge \square_{t,s,\ell} \wedge \heartsuit_{t,s}) &\implies \heartsuit_{t+1,s+\sigma} \end{aligned}$$

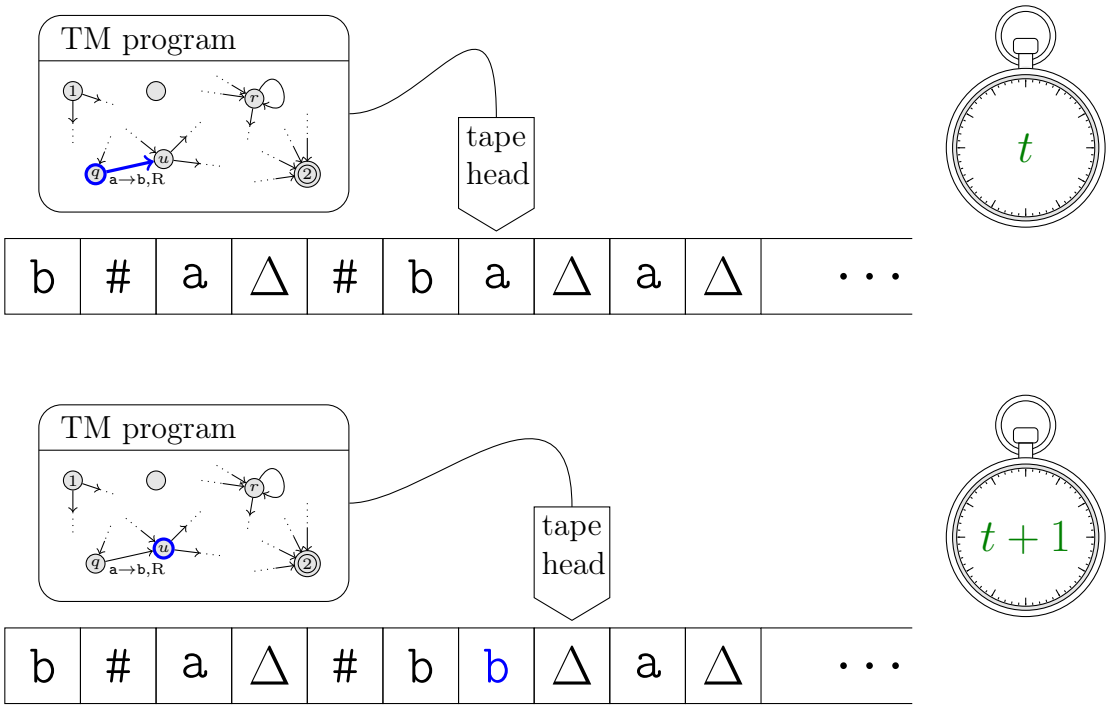
These can then be converted to CNF-clauses,

$$\begin{aligned} (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad &\vee \quad \bigcirc_{t+1,r}) \\ (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad &\vee \quad \square_{t+1,s,m}) \\ (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad &\vee \quad \heartsuit_{t+1,s+\sigma}), \end{aligned}$$

and then combined with conjunction:

$$\begin{aligned} & (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad \vee \quad \bigcirc_{t+1,r}) \\ \wedge \quad & (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad \vee \quad \square_{t+1,s,m}) \\ \wedge \quad & (\neg\bigcirc_{t,q} \vee \neg\square_{t,s,\ell} \vee \neg\heartsuit_{t,s} \quad \vee \quad \heartsuit_{t+1,s+\sigma}). \end{aligned}$$

Once this is done, these expressions (for all t, all s, all ℓ, and all transitions) are all combined using conjunction with the earlier expressions we have been constructing. The resulting expression ensures that the change from state t to state $t+1$ is in accordance with the appropriate transition (and that no other changes occur).



29.5 Conclusion

Finally,

$$\varphi_x \;:=\; \text{the conjunction of all the expressions we've made so far.}$$

The algorithm that takes input x and constructs φ_x as above,

$$x \longmapsto \varphi_x\,,$$

is our polynomial transformation from L to SATISFIABILITY.

We have constructed φ_x so that

$$x \in L \quad \text{if and only if} \quad \varphi_x \in \text{SATISFIABILITY}.$$

The construction can be done in polynomial time. This is lengthy, but routine, to prove. To gain insight on this, find upper bounds for the numbers of variables and clauses created, in terms of n and k, where the Verifier TM's time complexity is $O(n^k)$. □

Revision

Things to think about:

- *One detail has been omitted:*
  Our construction assumes that the computation accepts *at* time $T(n)$. What if the

TM accepts *before* time $T(n)$? We need to include some extra clauses in φ_x to deal with this. How?

next page

- Now that we *know* that SATISFIABILITY is NP-complete, how can we use it to show that other problems are NP-complete, without going to the same amount of trouble all over again?
- How do we show that SATISFIABILITY $\leq_P$ 3SAT ?

Reading:

- Sipser, section 7.4, pp. 304–311.
- M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., San Francisco, 1979. See especially §2.6.

Now that we *know* that SATISFIABILITY is NP-complete, how can we use it to show that other problems are NP-complete, without going to the same amount of trouble all over again?

Once you've established that SATISFIABILITY (SAT) is NP-complete, you can use this knowledge to show that other problems are NP-complete without going through the same amount of reduction work for each problem. This is achieved using the concept of "reduction chains" or "transitive reduction," based on the following principle:

If you can demonstrate a polynomial-time reduction from a known NP-complete problem (e.g., SAT) to another problem (let's call it Problem X), then you can conclude that Problem X is NP-complete without repeating the reduction process for Problem X.

Here's how this process works:

1. Start with SAT: You know that SAT is NP-complete, and you've already shown this through a reduction.

2. Choose a New Problem: Identify the problem you want to prove as NP-complete. Let's call it Problem X.

3. Prove the Reduction: Construct a polynomial-time reduction from SAT to Problem X. In other words, show that you can take any instance of SAT and transform it into an equivalent instance of Problem X in polynomial time.

4. Conclude NP-Completeness: Once you've successfully demonstrated the reduction from SAT to Problem X, you can now conclude that Problem X is NP-complete. This is because you've shown that SAT, a known NP-complete problem, can be reduced to Problem X, which implies that Problem X is at least as hard as SAT.

This method saves you from having to perform the full reduction process for each new problem you want to prove as NP-complete. Instead, you leverage the existing knowledge of SAT's NP-completeness to establish the NP-completeness of other problems efficiently.

Keep in mind that, for this method to work, the reduction from SAT to Problem X must be valid and polynomial-time. If you can successfully create this reduction, you can confidently add Problem X to the list of NP-complete problems without repeating the entire reduction process for each problem. This is why NP-completeness proofs often rely on a well-established set of known NP-complete problems and a transitive reduction process.

How do we show that SATISFIABILITY $\leq_P$ 3SAT?

To show that SATISFIABILITY (SAT) is polynomial-time reducible to 3SAT, you need to demonstrate that you can transform any instance of SAT into an equivalent instance of 3SAT in polynomial time. In other words, you need to design a polynomial-time reduction from SAT to 3SAT.

Here's a high-level explanation of how this reduction can be performed:

1. **Input**: Start with an arbitrary instance of SAT, which consists of a Boolean formula in conjunctive normal form (CNF).

2. **Transformation**: To reduce this instance to 3SAT, you need to modify the CNF formula in such a way that it remains equivalent (i.e., has the same set of satisfying assignments) but is now in 3SAT format, which means each clause contains exactly three literals.

3. **Clause Transformation**: For each clause in the original CNF formula, you may need to introduce additional variables and clauses to ensure that each clause contains exactly three literals. This may involve converting some clauses into 3SAT format while keeping the overall formula equivalent.

4. **Example**: To illustrate, if you have a clause with more than three literals, say (A ∨ B ∨ C ∨ D), you can convert it into multiple 3SAT clauses. For instance:

   - (A ∨ B ∨ X)
   - (¬X ∨ C ∨ Y)
   - (¬Y ∨ D ∨ E)
   - (¬E) (to ensure the original clause is satisfied)

Here, X, Y, and E are new variables introduced to create these 3SAT clauses. This transformation ensures that the original clause and the new clauses are equivalent, and each clause contains exactly three literals.

5. **Completing the Reduction**: Repeat this process for each clause in the original CNF formula and for each variable in the original formula if necessary. The goal is to construct an equivalent 3SAT formula.

6. **Polynomial Time**: Ensure that this transformation can be performed in polynomial time. The number of new variables and clauses introduced is bounded by a polynomial in the size of the original SAT instance.

7. **Equivalence**: Prove that the 3SAT formula you obtained is equivalent to the original SAT formula, meaning they have the same set of satisfying assignments. This ensures that a solution to the 3SAT instance corresponds to a solution to the original SAT instance.

Once you've completed this transformation, you've shown a polynomial-time reduction from SAT to 3SAT. This demonstrates that 3SAT is at least as hard as SAT, and it's a fundamental result in computational complexity theory. It implies that if you can efficiently solve 3SAT, you can also efficiently solve SAT, making 3SAT NP-complete.

NP-COMPLETE

① Problem is in NP
② Every language in NP has a polynomial time mapping reduction to the problem

Example problem: Show Language L is NP-complete.

method: Show that SAT reduces to L in polynomial time + L is in NP
** SAT ⇒ SATISFIABILITY

SAT $\leq_p$ L: → SAT
Show ② by showing that known NP-complete problem does have mapping reduction to L in polynomial time.
$X \leq_p K \leq_p L$ where X is any language in NP

There exist 2 polynomial time mapping reductions so it will still overall be polynomial time.

Can NP-complete problems be reduced to any NP problems?
Assuming $P \neq NP$,

$P \subseteq NP$ so if a NP-complete problem L could be polynomial time reduced to every other NP problem then L could be polynomial time reduced to a problem in P which implies that L is in P so P = NP

BUT that would contradict the assumption of $P \neq NP$ so NP-complete problems can't be reduced to any NP problem.

So when possible NP-complete problems is reduced from a known NP-complete problem ...

known_problem $\leq_p$ possible_problem

... then it demonstrate that possible NP-complete problem is also NP-complete.

THEOREM 13: If K is NP-complete & L is in NP, $K \leq_p L$ then L is in NP-complete

It is given that L is in NP so for L there exist a verifier that verifies in polynomial time for the possible solution.

H is a language in NP that also exist a verifier for H that verifies a possible solution for H in polynomial time.

$H \leq_p K$ by the NP-completeness of K.

It is given that $K \leq_p L$ so by transitivity of $\leq_p$, $H \leq_p L$
Hence, L is NP-complete.

EXAMPLE QUESTION: vertex-cover is NP-complete.

① vertex-cover is NP

certification: a set of valid k vertices that covers all edges in the graph
verification process: check if graph contains exactly k vertices and check if each edges of graph is connected to the vertices given from the certificate where for each edge (u, v) at least one of u or v is in the certificate
→ this is in polynomial time based on number of edges and vertices present in the graph

② mapping reduction to known NP-complete problem (3-SAT)
$3\text{-SAT} \leq_p$ vertex cover

Boolean expression φ in CNF with exactly 3 literals in each clause
1. construct graph G_φ + positive integer k_φ:
   - Graph G_φ vertices:
     - For each variable in 3-SAT instance, create 2 vertices: x, $\neg x$
     - For each clause in 3-SAT instance, create 3 vertices that represents the literal within the clause: m_1, m_2, m_3
   - Graph G_φ edges:
     - ↳ represents relationship between literals, clauses & variables
     - ↳ valid vertex cover that satisfy 3-SAT instance
     - (m_1, m_2), (m_1, m_3), (m_2, m_3)

- minimum vertex cover size in $G_\varphi \geq 2m+n$
  - m = number of clauses in 3-SAT
  - n = number of variables

- more edges added depending on structure of 3-SAT formula
  - edges connect vertices representing literals to clause positions
  - eg. literal : jth position of ith clause
    - ↳ if literal = x_k, edge (ij, x_k) added
    - ↳ if literal = $\neg x_k$, edge $(ij, \neg x_k)$ added

construction of graph G_φ ensure vertex cover size is tied to the truth assignment to variables
↳ satisfying assignment correspond to vertex covers of size exactly k_φ when $k_\varphi = 2m + n$

ii. φ is satisfiable if and only if G_φ have vertex cover of size $\leq k_\varphi$
(slide 79/82, page 205)

** reduction above is indeed polynomial time

*** NP-complete problems: ① SAT
② 3-SAT
③ vertex cover
④ independent set
⑤ clique

③ - ④ - ⑤
vertex cover $\leq_p$ independent set
vertex cover $\leq_p$ clique

Lecture 30

Proving NP-completeness

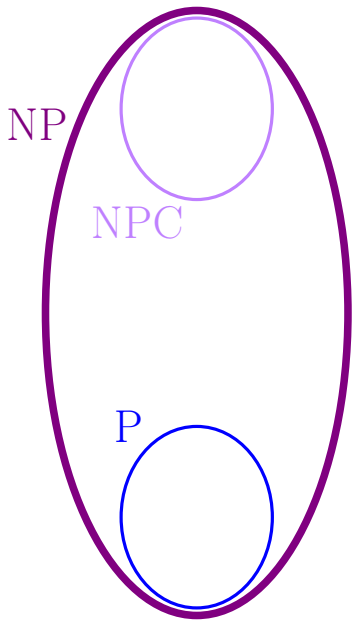
Now that we have proved that SATISFIABILITY is NP-complete, we can use this fact to help prove that other languages are NP-complete, using polynomial-time reductions.

We do this for 3-SAT, VERTEX COVER, INDEPENDENT SET and CLIQUE.

30.1 Language classes

We can now update our Venn diagram of the language classes P and NP to include the class of NP-complete languages, denoted by NPC in the diagram. So far, we can place one language in NPC, namely SATISFIABILITY.

If $P \neq NP$:



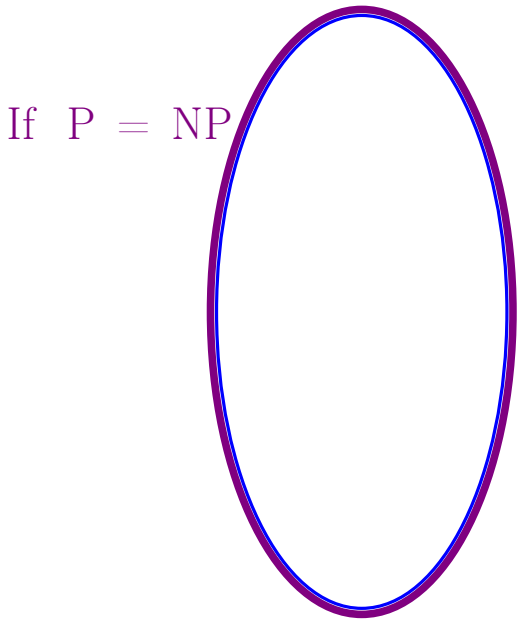
NP-complete:
SATISFIABILITY.

In NP, not known to be in P:
3-SAT,
HAMILTONIAN CIRCUIT,
3-COLOURABILITY,
VERTEX COVER,
INDEPENDENT SET, ...

GRAPH ISOMORPHISM,
INTEGER FACTORISATION, ...

In P:
2-SAT,
EULERIAN CIRCUIT,
2-COLOURABILITY,
CONNECTED GRAPHS,
SHORTEST PATH,
PRIMES,
Invertible matrices,
...,
All Context-Free Languages,
All Regular Languages.

If P = NP, the diagram is identical to the one we gave in Lecture 25 (p. 279). In this situation, every language in the class is also NP-complete, with two trivial exceptions.[1]



SATISFIABILITY, 3-SAT,
HAMILTONIAN CIRCUIT,
3-COLOURABILITY,
VERTEX COVER,
INDEPENDENT SET, ...

GRAPH ISOMORPHISM,
INTEGER FACTORISATION, ...

2-SAT,
EULERIAN CIRCUIT,
2-COLOURABILITY,
CONNECTED GRAPHS,
SHORTEST PATH,
PRIMES,
Invertible matrices,
...,
All Context-Free Languages,
All Regular Languages.

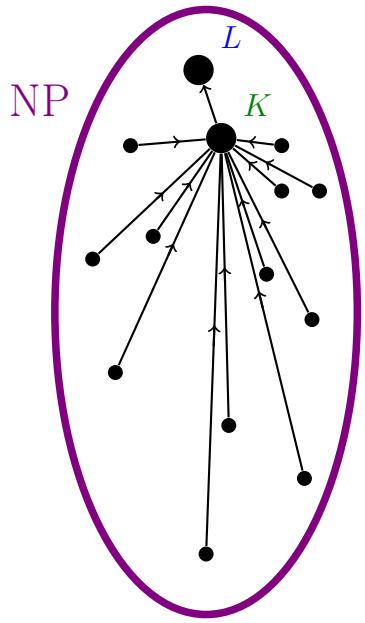
30.2 How to prove more NP-completeness results

Now that we have one NP-complete language, it is much easier to prove NP-completeness for many other languages.

Theorem 73
If K is NP-complete, L is in NP, and $K \leq_P L$, then L is NP-complete.

Proof.
We are given that L is in NP.
Let H be any language in NP.
Then $H \leq_P K$, by the NP-completeness of K.
Also, $K \leq_P L$ (which is given).
So, by the transitivity of $\leq_P$, we conclude that $H \leq_P L$.
Therefore L is NP-complete.
This theorem is illustrated in the next diagram. □

[1] What are those trivial exceptions, and why are they exceptional here?



So, to prove a language L is NP-complete, it is sufficient to

(a) show L is in NP, *and*

(b) show that another NP-complete language polynomial-time-reduces to L.

So, instead of providing an infinite family of polynomial-time reductions to reduce from everything in NP to SATISFIABILITY (as we did when proving the Cook-Levin Theorem in Lecture 29), we can now prove NP-completeness with just one reduction!

This approach didn't help us show SATISFIABILITY is NP-complete, since at that stage, we didn't know of any other NP-complete languages.

But now we can use SAT to show that other languages are NP-complete.

This approach was first taken by Richard Karp (1972) in the following paper.

- Richard Karp, Reducibility among combinatorial problems, in: R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations (Proceedings of a Symposium held March 20–22, 1972, IBM Thomas J. Watson Research Center, Yorktown Heights, New York)*, Plenum Press, New York, 1972, pp. 85–103.

30.3 Proving NP-completeness: 3-SAT

We first apply this approach to 3-SAT (defined on p. 271 in Lecture 25).

We can prove that it belongs to NP using the methods given in Lecture 25.

We can then prove that SAT $\leq_P$ 3-SAT.

This polynomial-time reduction works as follows.

Given a Boolean expression φ in CNF, each clause of φ is to be replaced by a clause, or clauses, of size 3, to do the same job.

The easiest clauses to handle are those that already have three literals! They are unchanged. Then, clauses of two literals are replaced by two clauses of three literals each using

a new variable for each such case (where that new variable appears nowhere else). We have seen this trick before, in the proof of Theorem 63 in Lecture 26. Clauses of one literal are each replaced by four clauses of three literals using a version of the same trick. Here are the replacements:

$$(x_1 \vee x_2 \vee x_3) \longmapsto \text{itself}$$

$$\begin{aligned}(x_1 \vee x_2) \longmapsto\ & (x_1 \vee x_2 \vee w_i) \\ & \wedge (x_1 \vee x_2 \vee \neg w_i)\end{aligned} \qquad \ldots \text{where } w_i \text{ appears nowhere else}$$

$$\begin{aligned}(x) \longmapsto\ & (x \vee w_{i1} \vee w_{i2}) \\ & \wedge (x \vee w_{i1} \vee \neg w_{i2}) \\ & \wedge (x \vee \neg w_{i1} \vee w_{i2}) \\ & \wedge (x \vee \neg w_{i1} \vee \neg w_{i2})\end{aligned} \qquad \ldots \text{where } w_{i1}, w_{i2} \text{ appear nowhere else.}$$

A different type of replacement is needed for clauses of four or more literals.

$$\begin{aligned}(x_1 \vee x_2 \vee x_3 \vee x_4) \longmapsto\ & (x_1 \vee x_2 \vee z_1) \\ & \wedge (\neg z_1 \vee x_3 \vee x_4)\end{aligned}$$

$$\begin{aligned}(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \longmapsto\ & (x_1 \vee x_2 \vee z_1) \\ & \wedge (\neg z_1 \vee x_3 \vee z_2) \\ & \wedge (\neg z_2 \vee x_4 \vee x_5)\end{aligned}$$

$\ldots etc,$

where each z_j appears nowhere else.

Then we must prove that the new expression so constructed is satisfiable if and only if the original expression is satisfiable, and that the construction is polynomial-time.

Once that is done, the proof that 3-SAT is NP-complete is complete.

3-SAT is a very handy NP-complete problem. It is a special case of SAT in the sense that 3-SAT $\subset$ SAT. The restriction to CNF expressions in which all clauses have *exactly* 3 literals seems quite strict. Superficially, 3-SAT may seem simpler than SAT. Doing polynomial-time reductions from 3-SAT to other languages is often easier than doing the reductions from SAT, because 3-SAT is simpler and more regular in structure. As a result, 3-SAT is more often used in NP-completeness proofs than SAT, even though it was SAT that started it all.

Our next NP-completeness proof is a case in point.

30.4 VERTEX COVER

We now show that VERTEX COVER is NP-complete. For its definition, see Lecture 25, p. 273.

We have seen that it's in NP.

To prove completeness, we show that

$$\text{3-SAT} \leq_P \text{VERTEX COVER}.$$

Given a Boolean expression φ in CNF with exactly 3 literals in each clause, we must show how to construct a graph G_φ and positive integer k_φ such that

$$\varphi \in \text{3-SAT} \quad \text{if and only if} \quad (G_\varphi, k_\varphi) \in \text{VERTEX COVER}$$

Suppose φ has

- variables $x_1, \ldots, x_n$,
- clauses $C_1, \ldots, C_m$.

We describe the construction of G_φ.
As with any graph, we start by specifying its vertices. It has:

- one vertex for each of the $2n$ literals of φ:

$$x_1, \neg x_1, \ldots, x_n, \neg x_n$$

- three vertices for each of the m clauses of φ:

$$C_{11}, C_{12}, C_{13}, \quad C_{21}, C_{22}, C_{23}, \ldots, C_{m1}, C_{m2}, C_{m3}$$

Then we define the edges of G_φ.

- Join each literal to its "partner" (i.e., its negation), which makes n edges:

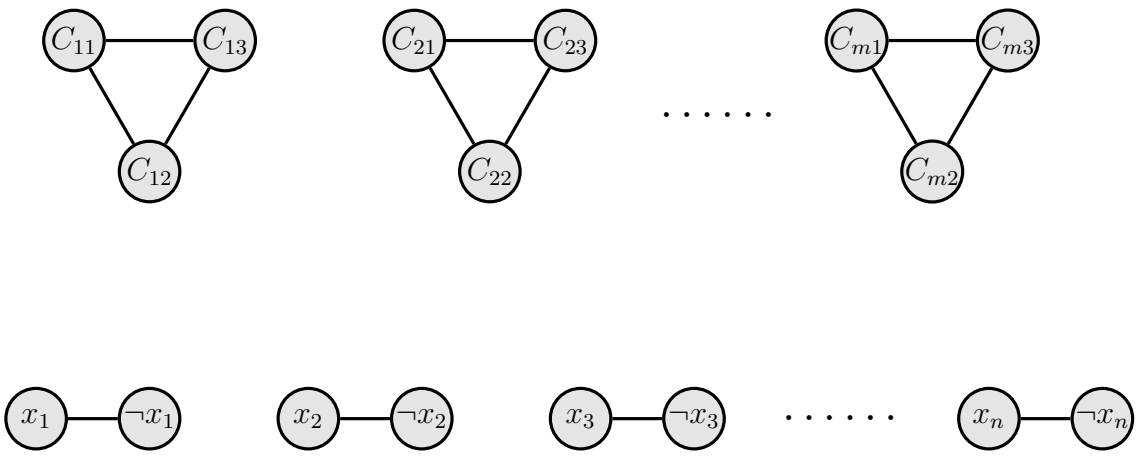
$$(x_1, \neg x_1), \ldots, (x_n, \neg x_n)$$

- Join up all vertices corresponding to each clause, which makes $3m$ edges:

$$(C_{11}, C_{12}), (C_{11}, C_{13}), (C_{12}, C_{13}), \ldots\ldots,$$

  so each clause is a separate triangle in G_φ.

The portion of G_φ that we have constructed so far is shown, for an example, in the next diagram.



Looking at the graph so far, how large must a vertex cover be?

We make the following observations.

Each "variable-edge" must be covered, by at least one vertex of the VC.

Each "clause-triangle" must be covered, by at least two vertices of the VC.

All variable-edges and clause-triangles are disjoint from each other.

So all vertex covers must have size $\geq 2m + n$.

Set $k_\varphi := 2m + n$.

This forces the vertex cover to have exactly one vertex from each variable-edge and exactly two vertices from each clause-triangle.

We resume describing the construction of G_φ. We come to the part of the construction that uses the detailed structure of φ, i.e., which literals appear in which clauses.

For each position in each clause:

- Add an edge from the vertex representing the clause position to the vertex representing the corresponding literal.
- So, for the literal in the j-th position of the i-th clause:
  - If the literal is x_k, then add the edge (C_{ij}, x_k).
  - If the literal is $\neg x_k$, then add edge $(C_{ij}, \neg x_k)$.

This completes construction of G_φ.

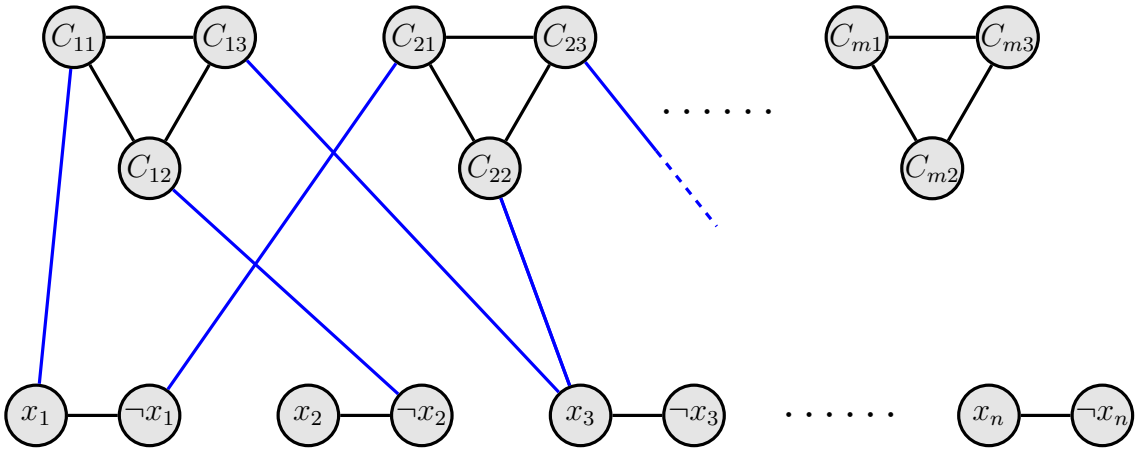
For example: if

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge \cdots$$

then

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge \cdots$$

The initial portion of G_φ is as follows.



For each variable-edge, the choice of which endvertex is in the vertex cover corresponds to the assignment of a truth value to that variable. A vertex being chosen should correspond to its corresponding variable being True.

The chosen literal covers all edges going from it up to the clauses.

So, if literal x_k is True and it appears in position j in clause C_i, then the edge (x_k, C_{ij}) is covered by x_k and does not need to be covered again in the clause-triangle for C_i .

So the vertex C_{ij} does not need to be in the vertex cover; the clause-triangle for C_i can be covered by its other two vertices.

So every clause containing a true literal is easily covered by two vertices.

Conversely, if a clause-triangle only has two vertices from the vertex cover, then the other vertex (not in the VC) gives the position of a literal which must be covered.

Therefore

The current truth assignment is satisfying
if and only if every clause has a true literal
if and only if the vertex cover only meets each clause-triangle twice
if and only if the vertex cover has size $\leq k_\varphi$.

So we have:

φ is satisfiable if and only if G_φ has a vertex cover of size $\leq k_\varphi$.

It remains to show that the reduction is polynomial time. This is fairly routine.

This concludes the proof that VERTEX COVER is NP-complete.

We now have three NP-complete problems:

SAT, 3-SAT, VERTEX COVER.

We saw in Lecture 26 (polynomial-time reductions) that

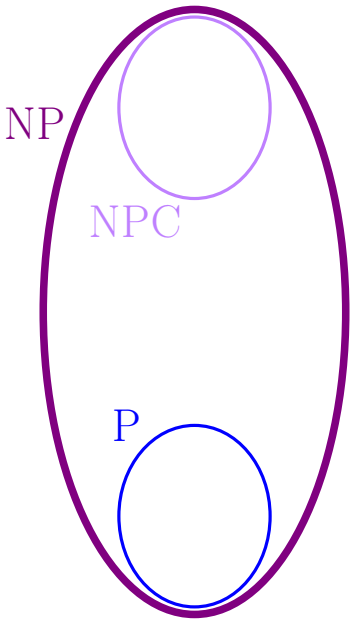
$$\begin{aligned} \text{VERTEX COVER} &\leq_P \text{INDEPENDENT SET} \\ &\leq_P \text{CLIQUE} \end{aligned}$$

so these two languages are NP-complete too.

✳ It is a good challenge to prove that 3-COLOURABILITY is NP-complete, by reduction from INDEPENDENT SET.

We now revise our Venn diagram.

If P $\neq$ NP:



NP-complete:
SATISFIABILITY, 3-SAT,
HAMILTONIAN CIRCUIT,
3-COLOURABILITY,
VERTEX COVER,
INDEPENDENT SET, ...

In NP, not known to be in P or NP-complete:
GRAPH ISOMORPHISM,
INTEGER FACTORISATION, ...

In P:
2-SAT,
EULERIAN CIRCUIT,
2-COLOURABILITY,
CONNECTED GRAPHS,
SHORTEST PATH,
PRIMES,
Invertible matrices,
...,
All Context-Free Languages,
All Regular Languages.

30.5 Implications of NP-completeness

Showing that a language is NP-complete does not make it go away! NP-complete languages are mostly motivated by real-world problems and the need to find practical solutions for them remains.

NP-completeness is strong *evidence* (but not *proof*) that you won't find an algorithm that is

efficient,
deterministic,
works in **all cases**, *and*
solves the problem **exactly**.

So you have several options:

a **slow** algorithm (exponential time)	not efficient
randomised algorithm	not deterministic
an algorithm for **special cases**	not all cases
approximation algorithm	not exact

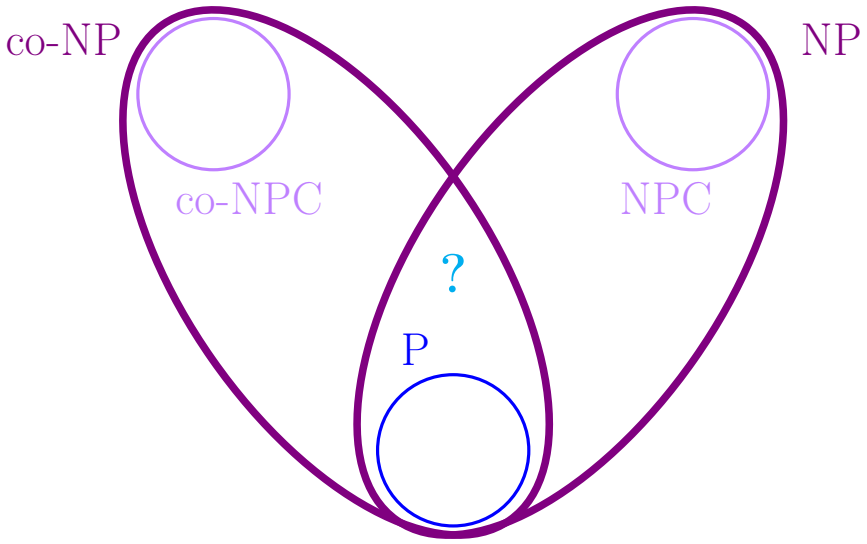
Which option to choose depends on the situation. Maybe the instances of your problem are not astronomically large, so that a smart exponential time algorithm might do the job. Maybe your problems are special cases of a more general NP-complete problem, with your special cases being easier to solve than the general case. Maybe you do not need a perfect solution; a good approximation might be sufficient.

Looking further into the future, some problems can be solved significantly more quickly on a quantum computer than on the kind of classical computers that are used (almost) ev-

erywhere today and which we have been studying in this unit.

Here is our final Venn diagram of language classes.

If $P \neq NP$ and $NP \neq$ co-NP:



The diagram highlights another open question: Does $P = NP \cap$ co-NP ?

There doesn't seem to be much difference, in practice, between these two language classes. For those who are interested, we discuss this (and much else) further in Lecture ω.

Revision

Things to think about:

- How to show that 4-SAT is NP-complete?
- How to show that 3-COLOURABILITY is NP-complete?
- What about the complexity of the following problem?

finding "vaccination set" that dominantly cover unvaccinated subgraph

VACCINATION

INPUT: Graph G, positive integers v, k.

QUESTION: Can you "vaccinate" v vertices of G so that all connected unvaccinated subgraphs have $\leq k$ vertices?

Answer: not polynomial

complexity depends on v and k values and graph structure

① polynomial
① fixed v and k
② v, k as input themselves
③ graph structure

Reading:

- Sipser, sections 7.4, 7.5.
- M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., San Francisco, 1979, §3.1. See especially §3.1.1 (3SAT) and §3.1.3 (VERTEX COVER).