Nondeterministic Turing machines and NP

1 The complexity class P

See also the video lecture recording: The complexity class P on Canvas.

So far we've seen various kinds of fancy Turing machines (allowing extra symbols, providing an extra tape, two-dimensional, ...). In each case, we've seen how to convert a fancy Turing machine M into a simple machine $\theta(M)$, in such a way that if M is polytime, then so is $\theta(M)$. This means that the question "is there a polytime machine that solves this problem?" doesn't depend on which kind of machine we use.

As before, we'll write Σ for our (input) alphabet.

Let f be a function from Σ^* to Σ^*. Saying that a Turing machine *computes* the function f means the following. For any word w, if the machine starts with just w on the tape and the head on the leftmost character (or just on a blank if $w = \varepsilon$), the machine eventually halts with $f(w)$ on the tape. For a given function f, we can ask: is there a polytime TM that computes it?

Let L be a language, i.e. a subset of Σ^*. Saying that a Turing machine *decides* the language L means the following. For any word w, if the machine starts with just w on the tape and the head on the leftmost character (or just on a blank if $w = \varepsilon$), the machine eventually returns True if $w \in L$, and False if $w \notin L$. For a given language L, we can ask: is there a polytime TM that decides it? A language thus corresponds to a *decision problem*, i.e. a problem where the answer is True or False.

The complexity class $\mathbf{P}$ is the set of all languages that can be decided in polytime. According to Polly, these are the decision problems that can be solved "fast".

A famous example is the set of prime numbers (represented in binary). Note here that the size of an input is the length of the bitstring, not the number itself. In 2002, it was shown to be in $\mathbf{P}$ by two computer science undergraduates (Kayal and Saxena) and their supervisor (Agrawal).

A machine is said to run in *exponential time* when its running time is $O(2^{n^k})$ for some $k \in \mathbb{N}$. The class of languages that can be decided in exponential time is called $\mathbf{EXP}$.

2 Representing data as words

This is all very well for problems involving words. But what if you want to study computational problems involving other kinds of data? For example, an *ordered pair* of words? Or a graph? Or an integer matrix? Or a database of student records?

In each case, we can devise a way of representing the data as a word. For example, for integer matrices, we can represent all the numbers in decimal, using a minus symbol for negative numbers, a comma to separate entries, and a newline character at the end of each line.

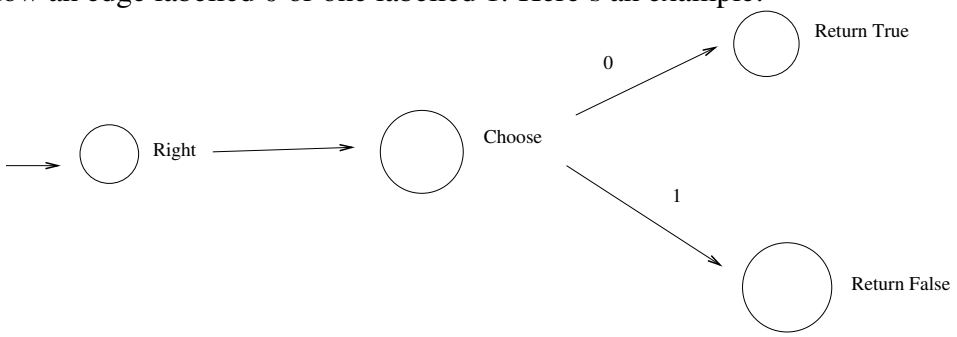
Of course this is just one way of representing an integer matrix as a word. You might invent another one to use instead. But I expect you will find that (a) the two representations differ in length by a constant factor and (b) you can convert in each direction using a polytime machine. So the choice of representation will not affect what counts as polytime.

In the same way, an ordered pair of words can be encoded as a word, or a graph, or a database of student records. In summary, any kind of data that can be written out as a finite amount of data can be encoded as a word.

3 Nondeterministic Turing machines

See also the video lecture recording: Nondeterministic Turing machines on Canvas.

Now we're going to look at a *nondeterministic* Turing machine, which can choose whether to follow an edge labelled 0 or one labelled 1. Here's an example:



Given a word w, we start the machine with just w on the tape and the head on the leftmost character. (It's on a blank symbol if $w = \varepsilon$.) The word is *acceptable* if the machine may return True, and *unacceptable* otherwise. The set L of acceptable words is the *language* of the machine.

A NDTM is *polynomial time* if there's M and C and k such that, for every word w of size $n \geqslant M$, when the machine starts with just w on the tape and the head on the leftmost character, the machine is *guaranteed* to terminate in time at most $C \times n^k$, regardless of the choices made.

The complexity class $\mathbf{NP}$ is the set of all languages that are given by a polytime NDTM. Since a TM is also a NDTM, we can see that $\mathbf{P} \subseteq \mathbf{NP}$. It's also the case that $\mathbf{NP} \subseteq \mathbf{EXP}$. To see this, suppose we have a NDTM that runs in time $C \times n^k$, for large enough n. Then to check all possible choices will take at most $2^{C \times n^k} \times C \times n^k$ steps, which is exponential.

Is it the case that $\mathbf{P} = \mathbf{NP}$? That is, can we convert a polytime nondeterministic machine M into a polytime deterministic machine N with the same language? Thus a word will be accepted by N iff it is acceptable to M.

This is an open question. The accepted hypothesis is that $\mathbf{P} \neq \mathbf{NP}$, but we don't know this for sure. We shall see why this is an important question.

4 Example:Sudoku

See also the video lecture recording: Sudoku on Canvas.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Some of you may have tried to solve a 3-Sudoku puzzle.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

It consists of 3×3 subgrids consisting of 3×3 cells. A solution is given in red. Every column, row and subgrid contains all the digits from 1 to 9. A general Sudoku puzzle is similar, but with $n \times n$ subgrids consisting of $n \times n$ cells; you fill it with numbers from 1 to n^2.

There are three computational problems associated with Sudoku.

- The *Sudoku checking* problem. Given a puzzle and a candidate solution, check that the solution is correct.
- The *Sudodu solvability* problem. Given a puzzle, decide whether it's solvable.
- The *Sudoku solution* problem. Given a puzzle, find a solution, or return Impossible.

The first two are decision problems, the last one is not.

The Sudoku checking problem is in $\mathbf{P}$. To check a candidate solution, there are three phases: checking the rows, checking the columns and checking the subgrids. Each row has n^2 entries. To check a row, we check each entry against every other entry, overall $n^2 \times n^2 = n^4$ comparisons (or a bit less). There are n^2 rows, so checking all the rows is $n^2 \times n^4 = n^6$ comparisons. The same for checking the columns and the subgrids. So we have $O(n^6)+O(n^6)+O(n^6) = O(n^6)$ comparisons.

The time taken for each comparison is proportional to the length of the numbers, so it is $O(\log n)$ steps. Overall we have $O(n^6) \times O(\log n) = O(n^6 \log n)$ steps, which is $O(n^7)$ steps. Note that we don't bother to write the base on $\log$, because up to a constant factor it doesn't matter.

Furthermore the size of a candidate solution is polynomial in the size of the puzzle. There are at most n^4 cells and each one is $O(\log n)$ bits.

Hence the Sudoku solvability problem is in $\mathbf{EXP}$. Given a puzzle, we can consider all possible candidate solutions, and check each one in polynomial time.

But it's also in $\mathbf{NP}$. Given a puzzle, we *guess* the solution bit by bit, which takes polynomially many steps, then check it, which takes polynomially many steps. A puzzle is acceptable to this machine iff it has a solution.

Thus if $\mathbf{P} = \mathbf{NP}$, then the Sudoku existence problem can be solved in polynomial time.

The solution problem (which is the practically useful one) is not a decision problem at all. But if the existence problem can be solved in polytime, then so can the search problem, by filling the bits one by one.

5 Example: Hamiltonian paths

See also the video lecture recording: Hamiltonian paths on Canvas.

Given a directed graph, a *Hamiltonian path* from vertex s to vertex t is a path that visits each vertex exactly once. So we have three problems:

- The *Hamiltonian path checking* problem. Given a graph with designated s and t, and a path from s to t, say whether it's Hamiltonian.
- The *Hamiltonian path existence* problem. Given a graph with designated s and t, is there a Hamiltonian path?
- The *Hamiltonian path search* problem. Given a graph with designated s and t, find a Hamiltonian path, or return Impossible.

The checking problem is in $\mathbf{P}$. The existence problem is in $\mathbf{NP}$, since we can guess the path, and its length is linear in the input size. The search is not a decision problem, but if the existence problem in $\mathbf{P}$, then the search problem can be solved in polytime, by filling the bits one by one.

6 Example: Factorization

See also the video lecture recording: Factorization on Canvas.

Given a number N we can check in polytime whether it's prime or composite. But if it's composite, this procedure will not give us the factors. We have three problems.

- The *factor checking problem*. Given N and c, say whether c is a factor of N. This can be done in polytime, using the long division algorithm.
- The *factor existence problem*. Given N and $k < N$, say whether N has a factor that is $\leqslant k$.

- The *factorization problem*. Given composite N, obtain a factor.

The checking problem is in $\mathbf{P}$, by implementing the long division algorithm. The existence problem is in $\mathbf{NP}$, as we guess a factor $\leqslant k$. The factorization problem is not a decision problem, but if the existence problem is in $\mathbf{P}$, then the factorization problem can be solved in polytime. To see this, we can use binary search to find the least prime factor. The number of iterations is linear (proportional to the length of N), and in each one we apply the polynomial algorithm for existence.

7 Example: SAT

See also the video lecture recordings: SAT and Checking on Canvas.

A *propositional formula* is built from atoms and connectives $\vee$ (disjunction, "or"), $\wedge$ (conjunction, "and") and $\neg$ (negation, "not"). For example, the formula $(\neg(q \vee p) \wedge r) \vee (p \wedge q)$. An *assignment* says whether each atom is true or false. For example the assignment $p = \mathrm{True}, \mathrm{q} = \mathrm{True}, \mathrm{r} = \mathrm{False}$. This is a *satisfying assignment* for the formula.

Again we get three computational problems.

- The *Formula satisfying assignment checking* problem. Given a formula ψ and an assignment, say whether it's a satisfying assignment for ψ.
- The *Formula-SAT* problem. Given a formula ψ, say whether it's satisfiable.
- The *Formula satisfying assignment search* problem. Given a formula ψ, find a satisfying assignment, or return Impossible.

Here are examples.

- $(p \vee \neg q \vee r) \wedge (\neg p \vee q \vee \neg r)$.
- $(p \vee \neg q) \wedge (p \vee q) \wedge (\neg p \vee \neg q)$

Are they satisfiable?

The checking problem can be performed in linear time. This uses an algorithm called "Shunting Yard" to parse the formula so that it can be easily evaluated.

Furthermore the size of a candidate assignment is linear in the size of the formula.

Therefore Formula-SAT is in $\mathbf{NP}$ because we can guess an assignment, which takes linear time, and then check whether it's a satisfying assignment, which takes linear time.

The search problem (which is the practically useful one) is not a decision problem at all. But if Formula-SAT can be solved in polytime, then so can the search problem, by filling the bits one by one.

SAT is useful for solving constraint problems. Let's see an example. Andy and Barbara each want to meet Chris between 15:00 and 17:00 for an hour, and they also want to meet each other for half an hour. The meeting times are 15:00, 15:30, 16:00 and 16:30. How do we turn this into an instance of SAT? We first give a list of 12 propositional variables:

AC(15:00)	Andy meets Chris at 15:00
AB(15:30)	Andy meets Barbara at 15:30

and so forth. Then we give a formula that expresses all the constraints. Scheduling the meetings is the same as finding a satisfying assignment.

8 The other definition of NP

Apart from the definition using NDTMs, there's another way of defining **NP**. A language L belongs to **NP** when there is a polynomial p and a polytime Turing machine M—called the *checking machine*—such that, for any word w (we write $|w|$ for its length), the following are equivalent:

- $w \in L$
- there is a word x of length $p(|w|)$ such that M accepts the ordered pair $\langle w, x \rangle$.

When this holds, we say that x *certifies* that $w \in L$.

Notice that the Sudoku solvability meets this definition. The word x is the candidate solution and its size is polynomial in $|w|$. (To be more precise, it's bounded by a polynomial p, but we can pad it out with 0's to give it length $p(|w|)$.) And, as we said at the outset, checking whether x is a solution to the puzzle w can be done in polyime.

Let's see why the two definitions are equivalent. (This is an outline argument, not a detailed proof.)

- Suppose that L is the language of a NDTM that, for an input w, runs in time $p(|w|)$. Then the number of choices is at most $p(|w|)$. Given a word w and suggested list of choices x of this length, we can check in time $p(|w|)$ whether this list leads to True being returned.
- Suppose that we have polynomial p and checking machine M. Then we can form a machine that, given w, guesses a word x of length $p(|w|)$ one bit at a time, which takes $p(|w|)$ steps, and then checks it in polytime. Altogether this is a polytime NDTM that recognizes L.

For each part, it's helpful to use auxiliary tapes. These can be removed using the methods we have learnt.

9 Completeness

See also the video lecture recording: NP Completeness on Canvas.

Let L and L' be languages. A *reduction* from L to L' is a function $f : \Sigma^* \to \Sigma^*$ such that for any bitstring x, we have $x \in L$ iff $f(x) \in L'$. It's a *polytime reduction* when this is done on a polytime machine.

The idea is that if we know how to decide membership of L', then the reduction gives us a way to decide membership of L. If we have a polytime reduction from L to L', and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$.

A language L is said to be **NP**-*hard* when every language in **NP** reduces to it in polytime. It is **NP**-*complete* when it is in **NP** and also **NP**-hard.

10 Completeness of Formula-SAT and 3CNF-SAT

An important result is that Formula-SAT is $\mathbf{NP}$-complete. Here is an outline of the proof. Suppose $L \in \mathbf{NP}$, i.e. there's a nondeterministic machine M that recognizes L in time bounded by the polynomial p. Given a word w, our task is to construct (in polytime) a formula that's satisfiable iff w is acceptable to M.

If w is acceptable, then there's a trace of machine configurations ending with Return true. There are at most $p(|w|)$ configurations, each of which uses at most $p(|w|)$ characters. So we have approximately $p(|w|)^2$ cell contents, as well as the states and the head positions. Each of these can be expressed by atoms. For example, we have an atom saying that at step 5, cell 17 contains 1; and another saying that at step 5 the head is over state 14; and another saying that at step 5, we're in state 17. Then we write down a load of constraints saying that the first configuration is the one that should be given by w, and each configuration (except the first) follows from its predecessor by following the rules of M, and the last constraint results in returning True.

This construction takes polytime, and the resulting formula is satisfiable iff w is acceptable. As required.

In fact, we can do better, and transform the formula into *3CNF* form. This refers to a formula that is a conjunction of clauses, each of which is a disjunction of three literals. (A *literal* is either an atom p or a negated atom, written $\overline{p}$.) For example:

$$(p \vee \overline{q} \vee r) \wedge (\overline{p} \vee q \vee \overline{s})$$

To sum up, 3CNF-SAT is an $\mathbf{NP}$-complete problem. This is called the *Cook-Levin theorem*. As for 2CNF, it is known to be in $\mathbf{P}$.

11 Other problems

Any problem that we can reduce 3CNF-SAT to in polytime must be $\mathbf{NP}$-hard. By this method, many problems have been shown to be $\mathbf{NP}$-complete, e.g. Sudoku solvability and the Hamiltonian path existence problem. Planning and scheduling problems are examples of $\mathbf{NP}$-complete problems that arise in AI and robotics.

As for factorization, the accepted hypothesis is that this is neither in $\mathbf{P}$ nor $\mathbf{NP}$-hard, but neither of these things is known for sure. The question is critical because RSA encryption is built on the assumption that factorizaion is hard. If it turns out to be easy, then RSA can be cracked.