LTCC Course: Graph Theory 2022/23
Lecture 3: Complexity and Algorithms

Jan van den Heuvel and Yani Pehova

21 November 2022

The graph theory textbooks do little or no algorithms, so for this lecture we have to go somewhere else.

The default textbook and source for anything algorithmic is: Thomas H. Cormen, Charles E. Leiverson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms* (1st–4th edition), MIT Press (1990–2022); `mitpress.mit.edu/9780262046305/introduction-to-algorithms/`.

An alternative (and lighter in weight) source is Herbert S. Wilf, *Algorithms and Complexity* (1st–2nd edition), Peters (1994–2003). The first edition (which is more than enough for us) can be downloaded for free from `www2.math.upenn.edu/~wilf/AlgComp3.html`.

The default book for complexity theory is still Michael R. Garey and David S. Johnson , *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman (1979) (ISBN: 0716710455). The only serious alternative we can think of is by Christos H. Papadimitriou, *Computational Complexity*, Addison Wesley (1994) (ISBN: 0201530821).

3.1 Computation

In the first two lectures of this course we mentioned algorithms a lot. Now we want to try to formalise what exactly 'an algorithm' to 'solve a problem' is. The simplest way to do this is to define a Turing Machine (TM).

Turing Machines

A (deterministic) TM has the following parts:

- an *alphabet* Σ, which is a finite set of symbols; one of which is #, called 'blank'; (A modern computer uses two symbols, but it is not exactly a TM; if you want to work with TMs it is usually convenient to use a larger alphabet.)
- a 2-*way infinite tape*, divided into *squares*, on each of which a symbol from Σ is written;
- a *tape head* which is always 'at' one square of the tape;
- a finite list Q of *states*, including the initial state q_0; and
- a *transition function* $\delta : Q \times \Sigma \to Q \times \Sigma \times \{\text{Left}, \text{Right}, \text{Halt}\}$.

Let $\mathcal{M}$ be a TM. At time 0, we always assume that 'the state of $\mathcal{M}$' is the initial state q_0. We call the square at which the tape head of $\mathcal{M}$ is initially positioned the 'zero square' (purely for convenience in discussion: the squares are *not* numbered in a way visible to $\mathcal{M}$!). A finite string I, called the *input*, is written on the tape, starting at the zero square and moving right, and all other squares on the tape are blank (i.e. # is written on them).

Now for each time $t \geq 0$ in succession, the following happens. Suppose $\mathcal{M}$ is in state q at time t, and suppose that the symbol σ is written on the square at which the tape head is positioned at time t. Let $\delta(q,\sigma) = (q', \sigma', m)$. Then the state of $\mathcal{M}$ at time $t+1$ is q' and the symbol σ' is written on the tape square where the tape head is positioned at time t (overwriting σ). The tape is otherwise not changed from time t to time $t+1$. Finally, the tape head is moved Left or Right according to m; or if $m =$ Halt, then the TM $\mathcal{M}$ halts, meaning no further action ever occurs. Observe that by definition, at any given finite time the tape head has visited only finitely many tape squares, so all but finitely many squares of the tape are still blank.

It will sometimes be useful to generalise this to a *multi-tape TM*, in which case we have a (finite) list of 2-way infinite tapes, each with a tape head which moves independently; the transition function reads all the tapes, updates all the tapes, and sets movement for each tape in each time step.

A good example of a TM is a modern computer with the hard disc replaced by a 2-way infinite tape (as above). Let's call this a *tape-computer*. The computer can do all kinds of complicated computation in between reading and writing the tape, but ultimately this computation takes only a finite time. (Because there are only so many possibilities for the internal memory; where we assume the computer avoids infinite loops in this internal computation!) And in fact the computation can be reduced to simply changing from one of a (very long!) list of states to another. You should believe from this example that a TM can do anything that a modern computer can do, if it has enough states.

In particular, one thing a tape-computer can do is *simulate* a TM, that is, if you input to a tape-computer a description of a TM $\mathcal{M}$ and its input I (written on the tape), then the tape-computer can work out for each time t what the result of running $\mathcal{M}$ on input I for time t will be. This makes it a *universal TM*; that's more or less the abstract definition of 'programmable computer'. So such things exist, if you use a TM with enough states. In fact, 'enough states' is not the colossal number that comes from this construction; 10 states is (easily) enough, though proving this is quite painful.

A given TM $\mathcal{M}$ with an input I can do one of two things: either it eventually halts, having *computed* the final contents of the tape (the finite non-blank string of symbols), or it never halts.

Problems

The standard way to describe a problem in theoretical computer science is as follows:

NAME-OF-PROBLEM-IN-CAPITALS
Input: *A description of the type of inputs that will be considered.*
Output: *Certain kind of output, usually related to the input.*

An example is:

INTEGER-ADDITION
Input: Two integers a and b.
Output: The sum $a + b$ of a and b.

An *instance* of a problem is a specific input of the prescribed type. So, an instance for INTEGER-ADDITION could be "$a = 514$ and $b = -6969696969$".

In fact, the above is more general than what we will study in these lectures. We will mostly look at so-called *decision problems*. These are problems in which the output should only be "Yes" or "No", depending on the answer of a certain question. Examples would be:

GRAPH-COLOURING
Input: An undirected graph G and an integer $K \geq 1$.
Question: Does there exist a vertex colouring of G with K colours?

GRAPH-K-COLOURING
Input: An undirected graph G.
Question: Does there exist a vertex colouring of G with K colours?

Instead of "an instance for which the answer is "Yes"", we will say "a true instance".

Note that there really is a difference between the decision problems GRAPH-COLOURING and GRAPH-K-COLOURING. The latter one only makes sense if the value of K is known or given from the outset. Once the value of K is decided or given, that K is used for every instance. In the first problem, K is part of the input and can vary from instance to instance.

It is important not to confuse the 'mathematical difficulty' of a problem with the 'algorithmic difficulty'. For example, INTEGER-ADDITION is mathematically 'trivial', but designing a Turing Machine which solves it, while not hard, is not trivial. On the other hand, consider the decision problem

FLT
Input: An integer $n \geq 1$.
Question: Does there exist a solution to $x^n + y^n = z^n$ with integers $x, y, z \geq 1$?

It's easy to see that the answer to this problem is 'Yes' if $n = 1$ or $n = 2$, but it is a famous and very difficult theorem of Wiles ('Fermat's Last Theorem') that for all larger n the answer is 'No'. So this problem is mathematically very far from trivial. Nevertheless, a machine which solves the problem is trivial to build: it must simply return 'Yes' if $n = 1, 2$ and otherwise 'No'. In fact (for many reasonable ways of inputting n) the machine doesn't even have to read the whole input and stops in constant time!

Words and Languages

Given an alphabet Σ (which we will assume is fixed from now on), a *word* is a finite sequence of symbols from the alphabet. The *length of a word* x, denoted $|x|$, is the number of symbols in it.
A special word is the *empty word*, often denoted by Λ. The empty word plays a role comparable to the empty set. In particular we have $|\Lambda| = 0$.

Given an alphabet Σ, we denote by Σ^* the set of all words using symbols from Σ. So we would have

$$\{2, x\}^* = \{\Lambda,\ 2,\ x,\ 22,\ 2x,\ x2,\ xx,\ 222,\ \ldots\}.$$

Given an alphabet Σ, a *language* $\mathcal{L}$ is a subset of Σ^*: $\mathcal{L} \subseteq \Sigma^*$.

In these lectures we don't want to spend time worrying about how to give the input to a machine. So we ignore most of the questions regarding "encoding" of instances in words over a certain alphabet. We just expect that it is possible for the problems we encounter; that there is some "natural encoding" of the instances of the problem we are looking at.

We require one property of any encoding of the instances of a problem Π: there can be no words $x \in \Sigma^*$ that encode more than one instance of Π.

The above description fits our experience with computers. Almost all modern computers do everything using just $\{0, 1\}$ as the alphabet. And even with that restricted alphabet there seem to be few restrictions of what can be represented to a modern computer. (Actually, there are major restrictions on what computers can deal with. In particular they can't work with binary expansions of real numbers like $\sqrt{2}$ and π.)

The only explicit encoding we will use is that if $\Sigma = \{0, 1\}$, then a non-negative integer N will be encoded as a binary number of length $\lfloor \log_2(N) \rfloor + 1$. All logarithms from now on will be assumed to be with base 2.

With the above convention on encoding in mind, we can define for a decision problem Π the corresponding language $\mathcal{L}_\Pi$:

$$\mathcal{L}_\Pi = \{x \in \Sigma^* \mid x \text{ is an encoding of a true instance of } \Pi\}.$$

And in fact, we will more or less identify the problem Π and the language $\mathcal{L}_\Pi$.

Despite the fact that we are glossing over 'encoding issues' in this course, you should be aware that they are sometimes very important. A good example is that it is possible to encode an integer either in 'unary' (n is represented by a string of n ones) or 'binary' (with which you are familiar), or indeed in many other ways. Consider the following two problems:

FACTORISE-UNARY
Input: An integer $n \geq 1$ presented in unary form.
Output: The prime factors of n in increasing order.

FACTORISE-BINARY
Input: An integer $n \geq 1$ presented in binary form.
Output: The prime factors of n in increasing order.

Now the first problem is easy to solve in polynomial time (polynomial in the length of the input), while the second is widely believed to be impossible to solve quickly. In fact, any quick algorithm would leave the world's private communication and banking wide open to criminals by making it easy to break the RSA system. The difference is that when n is input to the first problem, the length of the input is n, whereas when it is input to the second problem, the length of the input is $\lceil \log_2 n \rceil$. So in the first problem we are 'allowed' to take a much longer time to factorise n than in the second. Similarly (although this is more obviously 'cheating') we could define a representation of a graph in which we require that the vertices are given in colour order from some optimal proper vertex colouring. The problem CHEATING-GRAPH-COLOURING, which takes this representation together with an integer $K \geq 1$ as input and outputs 'Yes' if there exists a proper k-colouring of the graph, is also easy to solve, even though the problem GRAPH-COLOURING defined above, whose input is a graph in 'adjacency matrix form' together with an integer $K \geq 1$, is widely believed not to have any quick algorithm.

Solving problems

We say a given machine $\mathcal{M}$ *decides* a language $\mathcal{L}$ if $\mathcal{M}$ halts on any input $x \in \Sigma^*$, answering 'Yes' if $x \in \mathcal{L}$ and 'No' if $x \notin \mathcal{L}$. We'll also be interested in *accepting* $\mathcal{L}$, which is the weaker condition that $\mathcal{M}$ halts with 'Yes' if and only if $x \in \mathcal{L}$ (it may either answer 'No' or fail to halt if $x \notin \mathcal{L}$). If $\mathcal{L}$ is the language we get from a given decision problem (with our choice of input encoding), then we sometimes just say that $\mathcal{M}$ solves the problem. Observe that, by definition, if $\mathcal{M}$ decides a language at all, then it decides just one language.

Traditionally, Turing Machines that are supposed to solve problems have two special states q_Y and q_N. If the machine gets to one of those states, then it always halts. The state q_Y indicates the answer 'Yes'; q_N indicates 'No'. From now on we will also follow this convention.

3.2 Decidability

It's easy to see that there are languages which are not decided by any TM: there are a countably infinite number of possible TMs (for each number of states, there are only finitely many possibilities, and a countable union of finite sets is countable). But there are uncountably many languages. The power set of Σ^*, which is a countably infinite set, is the set of all languages. But are there any 'interesting' such languages? In particular, there are only countably many languages given by a decision problem that has a finite description (for the same reason as TMs). Is any decision problem with finite description undecidable? The following theorem answers this question.

Theorem 1 (Turing, 1936)**.**
Fix a way of encoding the pair $(\mathcal{M}, I)$ where $\mathcal{M}$ is a TM and I an input. Then the language $\mathcal{L}_{Halting}$, consisting of all pairs $(\mathcal{M}, I)$ where $\mathcal{M}$ halts eventually on input I, is undecidable.

Proof. Suppose to the contrary there is some TM $\mathcal{H}$ which decides $\mathcal{L}_{\text{Halting}}$. We construct a TM $\mathcal{P}$ which does the following:

1. For input x, check if x is a string representing a TM $\mathcal{M}$. Halt if not.
2. Simulate $\mathcal{H}$ with input $(\mathcal{M}, x)$.
3. If $\mathcal{H}$ halts in its 'Yes' state, then go into an infinite loop; if $\mathcal{H}$ returns its 'No' state then halt.

To justify that this is all possible, observe that (if you were given $\mathcal{H}$) it would be easy to program the rest on a tape-computer (It's not all that hard to do 'by hand', in fact. Try it as an exercise).

Now let p be the string representing the TM $\mathcal{P}$. The result of running $\mathcal{P}$ with input p is a contradiction. $\square$

The Halting Problem has connections with areas of mathematics that at first sight having nothing to do with algorithms. For instance, in 2016 Yedidia and Aaronson constructed a Turing Machine $\mathcal{R}$ with some input I such that $\mathcal{R}$ halts with input I if and only if the Riemann hypothesis is false. So if the Halting Problem was decidedable, then the pair $(\mathcal{R}, I)$ would give us a machine-checkable test to see if the Riemann hypothesis is true or false.

The Halting Problem is a fairly natural problem to want to solve (at least this is better than knowing undecidable languages exist by counting). But one can do better: Matiyasevich, in his 1970 PhD thesis, building on previous work of Davis, Putnam and Robinson, showed that the problem of whether a given Diophantine equation (a polynomial in many variables with integer coefficients) has integer solutions is undecidable. This problem was and is a central concern of number theory. Note that this does not 'show humans are better than computers'; a computer can simply check by brute-force all finite strings of symbols to see whether they constitute a valid proof, so anything a human can prove about (say) Diophantine equations a computer can in principle also prove.

The Church-Turing Thesis is the (non-mathematical!) assertion that any physically possible form of computation can be simulated by Turing Machines. There is no known counterexample. (Quantum computing, if it turns out to be physically possible, may give one, but is not known to give a counterexample to a much stronger statement which asserts that TMs are also at worst polynomially slower than any physically possible form of computing. TMs can simulate quantum computers, though they run exponentially slower using the best known algorithms.)

3.3 Order of Functions

In order to write simple comparisons, for functions $f, g : \mathbb{N} \to \mathbb{R}$, we define the following order comparisons:

– $f(n) = O(g(n))$, pronounced "*$f(n)$ is big oh $g(n)$*", means that there are constants $C > 0$ and $N \geq 1$ so that for all $n \geq N$ we have $|f(n)| \leq C|g(n)|$.

– $f(n) = o(g(n))$, pronounced "*$f(n)$ is little oh $g(n)$*", or "*... small oh ...*", means that for every constant $c > 0$ there is an $N_c \geq 1$, so that for all $n \geq N_c$ we have $|f(n)| \leq c|g(n)|$.

3.4 Polynomial Problems

A language $\mathcal{L} \subseteq \{0,1\}^*$ is *decided in polynomial time by a Turing Machine* $\mathcal{M}$ if $\mathcal{M}$ decides $\mathcal{L}$, and there is a polynomial $p(n)$ such that for every $x \in \{0,1\}^*$, $\mathcal{M}$ halts after at most $p(|x|)$ steps.

In the exercises you will prove that $f(n) \leq p(n)$ for some polynomial $p(n)$ of degree d is equivalent to $f(n) = O(n^d)$. So we can rephrase the definition above to:

A language $\mathcal{L}$ is *decided in polynomial time by a Turing Machine* $\mathcal{M}$ if $\mathcal{M}$ decides $\mathcal{L}$, and there is a positive integer d such that for every word x the machine $\mathcal{M}$ halts after $O(|x|^d)$ steps.

Similarly, we say a language $\mathcal{L}$ is *accepted in polynomial time by a Turing Machine* $\mathcal{M}$ if $\mathcal{M}$ accepts $\mathcal{L}$, and there is a positive integer d such that for every word $x \in \mathcal{L}$ the machine $\mathcal{M}$ halts after $O(|x|^d)$ steps.

The class P is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ for which there exists a Turing Machine that decides $\mathcal{L}$ in polynomial time.

Since there is a difference between 'deciding a language' and 'accepting a language', you should wonder if the definition of P would differ if we would use accepted instead of decided. In fact, this would not be the case.

Theorem 2.
Let Q be the class of all languages for which there exists a Turing Machine that accepts that language in polynomial time. Then $\mathrm{Q} = \mathrm{P}$.

Proof. We easily have $\mathrm{P} \subseteq \mathrm{Q}$, since if a language can be decided in polynomial time, then it can certainly be accepted in polynomial time. Now take a language $\mathcal{L}$ in Q. Then there is a Turing Machine $\mathcal{M}$ and a polynomial $p(n)$, so that for every word $x \in \mathcal{L}$ the Turing Machine halts in the 'Yes' state q_Y after at most $p(|x|)$ steps, while for a word $x \notin \mathcal{L}$ the Turing Machine either halts in the 'No' state q_N or doesn't halt at all.

Now we design a new Turing Machine $\mathcal{M}'$ as follows: For an input $x \in \{0,1\}^*$ it determines $|x|$ and calculates $p(|x|)$. It then simulates the action of $\mathcal{M}$ with input x for at most $p(|x|)$ steps, or less if $\mathcal{M}$ halts earlier. After that, $\mathcal{M}'$ inspects the outcome of the operations of $\mathcal{M}$. If $\mathcal{M}$ halted in its 'Yes' state, the word x is accepted; if $\mathcal{M}$ halted in its 'No' state or had not halted yet after $p(|x|)$ steps, the word x is rejected. This means that the Turing Machine $\mathcal{M}'$ decides the language $\mathcal{L}$.

The one remaining issue is if $\mathcal{M}'$ works in polynomial time. That is, can we determine $|x|$ and $p(|x|)$, and simulate $\mathcal{M}$ while simultaneously keeping track of the number of steps we use, without losing the polynomiality of the calculations? It's not so hard to show that this can indeed be done involving at most a polynomial factor extra. We skip the details. In essence, we simply first have to run once through x to determine $|x|$, then calculate $p(|x|)$ 'somewhere else' on the tape, and then while simulating $\mathcal{M}$ increase some 'step counter' after each step of $\mathcal{M}$ which is also written down 'somewhere else' on the tape.

The above shows that $\mathcal{M}'$ is a Turing Machine that accepts or rejects words in $\mathcal{L}$, and does so in polynomial time. We have proved that every language $\mathcal{L}$ in Q is also in P, and hence are done. $\square$

Because of the close relation between languages and decision problems, we also say about decision problems that they are in P. More generally, in the rest of this lecture we are going to be interested only in languages that happen to come from decision problems (with a specified encoding), and we will tend to refer to a language as being 'the same' as the decision problem with the specified encoding. We are generally going to ignore details, such as how we actually do encoding, so we will often simply state the decision problem without mentioning the encoding at all. Generally this laziness doesn't make any significant difference, but on occasion, particularly representing integers, it can make a difference. For us, integers will be written in binary. (To see why this is importantly different to unary, think about the FACTORISATION problem mentioned in the last notes. It doesn't matter whether the base is 2 or some larger integer though.)

3.5 Nondeterministic Turing Machines

A *nondeterministic Turing Machine* has all the elements of a normal Turing Machine, with one exception. Instead of a transition function, we have a *multivalued transition mapping*

$$\delta' : (Q \setminus \{q_Y, q_N\}) \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{\text{Left}, \text{Right}, \text{Halt}\}) \setminus \varnothing.$$

(Here for set X, $\mathcal{P}(X)$ denotes the *power set* of$X$, i.e. the collection of all subsets of X.)

When using a nondeterministic Turing Machine, at each step the transition is one of the elements of the transition mapping. We'll call such steps *allowed steps* or *allowed transitions.*

The actual step taken at each stage is nondeterministic in the sense that we don't know how the machine decides which of the allowed steps to choose. This is not the same as saying the steps are chosen at random. We can imagine that there is a little 'daemon' housed in the machine who decides which of the allowed steps to take, but we have no clue how this daemon comes to its decisions, if it always will make the same decisions in the same situation, and so on.

A word $x \in \Sigma^*$ is *accepted* by a nondeterministic Turing Machine $\mathcal{N}$ if, when given x as input, there is a finite sequence of allowed steps so that $\mathcal{N}$ halts in the 'Yes' state q_Y

A nondeterministic Turing Machine $\mathcal{N}$ *accepts a language* $\mathcal{L} \subseteq \Sigma^*$ if for all $x \in \Sigma^*$ we have that $\mathcal{N}$ accepts x if and only if $x \in \mathcal{L}$.

Again we have the situation that if a nondeterministic Turing Machine $\mathcal{N}$ accepts the language $\mathcal{L}$, then we don't know exactly what will happen if the input is an element $x \notin \mathcal{L}$. The machine may end after a finite number of steps in the 'No' state q_N. Or it may not halt at all. And because of the nondeterministic nature, sometimes the same input $x \notin \mathcal{L}$ may halt in state q_N, while other times that input may lead to a non-halting calculation.

But in fact, the situation is also much more complicated for words $x \in \mathcal{L}$ in the accepted language. We only know that there is at least one possible sequence of allowed steps that make $\mathcal{N}$ halt in the 'Yes' state q_Y. For that same input $x \in \mathcal{L}$, there may be many other sequences of allowed steps for which the machine halts in the 'No' state or doesn't halt it all.

A language $\mathcal{L} \subseteq \{0,1\}^*$ is *accepted in polynomial time by a nondeterministic Turing Machine* $\mathcal{N}$ if there is a polynomial $p(n)$ such that for every $x \in \mathcal{L}$, there exists a sequence of at most $p(|x|)$ allowed steps so that $\mathcal{N}$ halts in the 'Yes' state q_Y.

The class NP is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ for which there exists a nondeterministic Turing Machine that accepts $\mathcal{L}$ in polynomial time.

Proposition 3.
We have P $\subseteq$ NP*.*

Proof. A deterministic Turing Machine is just a special kind of nondeterministic Turing Machine (where in every situation there is only one allowed step). So we get that the set of all languages $\mathcal{L}$ that can be accepted by a deterministic Turing Machine in polynomial time is a subset of NP. But in Theorem 2 above we proved that the set of all languages that can be accepted by a deterministic Turing Machine is polynomial time is equal to P. □

3.6 Turing Machines as Verifiers

In this section we consider (deterministic) Turing Machines that 'verify' membership in a language. The idea is that to convince somebody that, say, a certain graph is 3-colourable, the easiest way would be to provide a 3-colouring of the graph. All the other party (who is sceptical about the 3-colourability of the graph) then has to do, is checking if the given colouring of the graph indeed is a proper colouring using at most three colours. The given 3-colouring is used as a 'certificate' of the fact that the graph is 3-colourable.

Note that in the above example, if the graph is not 3-colourable, then no certificate is possible that will fool somebody in thinking the graph is 3-colourable, provided their checking procedure is up to the job.

The class NP' is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ such that there is a deterministic Turing Machine $\mathcal{M}$ and polynomials $p_1(n)$, $p_2(n)$, such that:

- for all $x \in \mathcal{L}$ there is a *certificate* $y \in \{0,1\}^*$ such that
  - $|y| \leq p_1(|x|)$, and
  - $\mathcal{M}$ accepts the combined input 'x, y', halting after at most $p_2(|x| + |y|)$ steps;
- for all $x \notin \mathcal{L}$, there is no $y \in \{0,1\}^*$ so that $\mathcal{M}$ accepts 'x, y'.

Theorem 4.
We have NP $=$ NP$'$*.*

Sketch of Proof. If $\mathcal{L}$ is a language in NP, then there is a nondeterministic Turing Machine $\mathcal{N}$ that accepts every word $x \in \mathcal{L}$ in polynomial time. I.e. giving $x \in \mathcal{L}$ as the input to $\mathcal{N}$, there is a polynomial number of allowed steps that lead to the 'Yes' state q_Y of $\mathcal{N}$. So now consider the sequence of allowed steps the machine $\mathcal{N}$ has taken to reach q_Y. This sequence has at most polynomial length. And hence this sequence can be used as a certificate for a deterministic Turing Machine $\mathcal{M}$, a machine that simulates the nondeterministic Turing Machine $\mathcal{N}$. But instead of having multiple transitions, it consults the certificate to decide deterministically what its next step will be. So this machine will accept a word $x \in \mathcal{L}$, provided the certificate y is indeed a valid description of the allowed sequence of $\mathcal{N}$ leading to the 'Yes' state of $\mathcal{N}$.

By the definition of 'acceptance of a word' for a nondeterministic Turing Machine, if $x \notin \mathcal{L}$, then no allowed finite sequence leading to the 'Yes' state of $\mathcal{N}$ is possible, hence the deterministic Turing Machine $\mathcal{M}$ will never accept $x \notin \mathcal{L}$, whatever certificate y we give it.

We've shown that we can build a deterministic Turing Machine which is a verifier for words in $\mathcal{L}$. This proves NP $\subseteq$ NP$'$.

There are many ways to prove NP$'$ $\subseteq$ NP. Here is one possibility. Suppose $\mathcal{L}$ is a language in NP$'$, and let the deterministic Turing Machine $\mathcal{M}$ and the polynomials $p_1(n), p_2(n)$ be as described in the definition of the class NP$'$. We now built a nondeterministic Turing Machine $\mathcal{N}$

that accepts words $x \in \mathcal{L}$ as follows. Given input x, the machine first determines $|x|$ and calculates $p_1(|x|)$. It then goes into a nondeterministic mode where it generates a sequence $z \in \{0,1\}^*$ of length at most $p_1(|x|)$. This nondeterministic process must be set up so that any sequence from $\{0,1\}^*$ of length at most $p_1(|x|)$ is possible as an outcome, which is easy. Once that is done, $\mathcal{N}$ simulates all the steps $\mathcal{M}$ would have done with the combined input 'x, z'. (This stage is completely deterministic again.) From the definition of NP$'$, if $x \in \mathcal{L}$, there is a certificate y of length at most $p_1(|x|)$ so that $\mathcal{M}$ will accept the combined input 'x, y'. Since there is a sequence of allowed steps for $\mathcal{N}$ that will generate that y, and then do the calculations $\mathcal{M}$ would have done to verify 'x, y', this nondeterministic Turing Machine will accept exactly the elements in $\mathcal{L}$.

It is not so hard to convince oneself that the nondeterministic Turing Machine $\mathcal{N}$ can do all its work in polynomial time as well. $\square$

3.7 Complements of Languages and Decision Problems

In terms of decidability the following decision problems are equivalent:

GRAPH-3-COLOURING
Input: An undirected graph G.
Question: Does there exist a vertex colouring of G with 3 colours?

GRAPH-NON-3-COLOURING
Input: An undirected graph G.
Question: Does there not exist a vertex colouring of G with 3 colours?

But in terms of complexity theory, these questions are quite different. We've already seen that GRAPH-3-COLOURING is in NP, since there exists an easily verifiable certificate (a 3-colouring) that will convince us a certain instance is a positive one. But suppose we want to convince the other party that a certain graph is a positive instance of GRAPH-NON-3-COLOURING. It's not obvious what a good (i.e. polynomial time checkable) certificate that convinces us a graph is not 3-colourable should look like.

For a language $\mathcal{L} \subseteq \{0,1\}^*$, we define $\mathcal{L}^c = \{0,1\}^* \setminus \mathcal{L} = \{x \in \{0,1\}^* \mid x \notin \mathcal{L}\}$. If C is a class of languages, then coC is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ for which $\mathcal{L}^c \in$ C.

From the classes we've seen so far we obtain coP and coNP.

Following our identification of languages and decision problems, we can also look at the complement classes of decision problems. So if Π is the GRAPH-3-COLOURING decision problem above, then the corresponding language is:

$$\mathcal{L}_\Pi = \{x \in \Sigma^* \mid x \text{ is an encoding of a 3-colourable graph}\}.$$

So formally, the complement language should be:

$$\begin{aligned}\mathcal{L}^c_\Pi = \{x \in \Sigma^* \mid\ & x \text{ is not a proper encoding of any graph,} \\ & \text{or } x \text{ is the encoding of a graph that is not 3-colourable}\}.\end{aligned}$$

But this is not really the natural way to think about it. So we usually just ignore the possibility that x is not a proper encoding of any graph, and concentrate on the words that represent graphs, but are not true instances of the original decision problem. One of the arguments that this is not really a major issue is that it is easy to check if a certain word is a proper encoding of a graph, and that the real work is in testing the decision problem. Of course, if we are dealing with decision problems where it is not so trivial to check if a word represents one of the instances, then we should be much more careful and precise.

With this convention in mind, we allow ourselves to say that

$$\text{GRAPH-NON-3-COLOURING} = \text{co-GRAPH-3-COLOURING}.$$

Proposition 5.
We have the following relations: $\mathrm{P} = \mathrm{coP}$, $\mathrm{P} \subseteq \mathrm{NP}$, *and* $\mathrm{P} \subseteq \mathrm{coNP}$.

Hence we have $\mathrm{P} \subseteq (\mathrm{NP} \cap \mathrm{coNP})$.

Proof. The fact that $\mathrm{P} = \mathrm{coP}$ follows from the definition of P as all languages $\mathcal{L}$ that have a corresponding Turing Machine $\mathcal{M}$ that *decides* the language in polynomial time. Such a machine determines both if $x \in \mathcal{L}$ or $x \notin \mathcal{L}$, and does so in all cases in polynomial time.

The fact that $\mathrm{P} \subseteq \mathrm{NP}$ is Proposition 3. By an analogous argument we obtain $\mathrm{coP} \subseteq \mathrm{coNP}$ and hence $\mathrm{P} \subseteq \mathrm{coNP}$. □

It is unknown if P is equal to $\mathrm{NP} \cap \mathrm{coNP}$ or not.

Similarly, it is unknown if NP is really larger than P or not. This last question is one of the *Clay Millennium Problems*, and an answer is worth a million dollars. (See `www.claymath.org/millennium-problems/`.)

3.8 Polynomial Reducibility and Completeness

Until now we mostly looked at Turing Machines as abstract constructions to decide languages (equivalently: to 'solve' decision problems). Now we also start to consider Turing Machines as abstract constructions to compute functions.

Given a function $f : \{0,1\}^* \to \{0,1\}^*$, we say that a Turing Machine $\mathcal{M}$ *computes* f if for every word $x \in \{0,1\}^*$, if $\mathcal{M}$ is given the input x, then $\mathcal{M}$ halts after finite time, and once $\mathcal{M}$ is halted, the string that remains on the tape (the *output*) is the word $f(x)$.

Let $\mathcal{L}_1, \mathcal{L}_2$ be two languages. We say that $\mathcal{L}_1$ *is polynomial-time reducible to* $\mathcal{L}_2$, with the notation $\mathcal{L}_1 \leq_P \mathcal{L}_2$, if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$, a Turing Machine $\mathcal{M}$, and a polynomial $p(n)$ such that

- for all words x we have: $x \in \mathcal{L}_1 \iff f(x) \in \mathcal{L}_2$;
- the Turing Machine $\mathcal{M}$ computes f; and
- for all words x, $\mathcal{M}$ computes $f(x)$ in at most $p(|x|)$ steps.

Let C be a class of languages. A language $\mathcal{L}$ is *C-complete* if

- $\mathcal{L}$ is in C; and
- for all $\mathcal{L}' \in \mathrm{C}$ we have $\mathcal{L}' \leq_P \mathcal{L}$.

The way to interpret is that languages $\mathcal{L}$ that are C-complete for some class C are the 'hardest' problems in C: If there exists a polynomial time Turing Machine to decide $\mathcal{L}$, then *every* language in C can be decided in polynomial time.

Regarding the classes P and NP, we have the following easy facts.

Observation 6.
For any two languages $\mathcal{L}_1, \mathcal{L}_2$, if $\mathcal{L}_1$ is in P and $\mathcal{L}_2 \leq_P \mathcal{L}_1$, then $\mathcal{L}_2$ is in P.

If there exists a language $\mathcal{L}$ so that $\mathcal{L}$ is NP-complete and $\mathcal{L}$ is in P, then $\mathrm{P} = \mathrm{NP}$.

Proposition 7.
Suppose $\mathcal{L}_1$ is an NP-complete language. If $\mathcal{L}_2$ is a language so that $\mathcal{L}_2$ is in NP and $\mathcal{L}_1 \leq_P \mathcal{L}_2$, then $\mathcal{L}_2$ is NP-complete.

The proof is an exercise.

Somewhat less obvious is the following, also an exercise.

Proposition 8.
Every non-trivial language in P is P-complete.

Since all non-trivial languages in P are P-complete, we have a long list of decision problems that are P-complete. But it's far less obvious that there exist NP-complete problems. The first decision problem that was shown to be NP-complete is the so-called SATISFIABILITY problem of Boolean logical expressions, by Cook in 1971. Levin, about the same time and independently, proved a similar result. (Levin showed that certain 'search problems' are 'universal'; it is not too hard to deduce Cook's result from this and vice versa.)

A Boolean formula has *variables* $x_1, \dots, x_n$. In addition the symbols $\neg$ (Boolean NOT), $\wedge$ (AND), $\vee$ (OR) and brackets are used. So an example of a Boolean formula is $(\neg x_1 \vee x_2) \wedge (x_3 \vee x_2)$.

SATISFIABILITY is simply the problem: given a Boolean formula, is there a satisfying instance (an assignment of True and False to the variables such that the formula evaluates to True)?

Typically SATISFIABILITY is stated in terms of specifically Conjunctive Normal Form formulae. This means (to save space) we write $\overline{x_i}$ rather than $\neg x_i$, we write *clauses* which are sets of variables and their negations, written for example $(\overline{x_1}, x_2)$ which evaluates to True if at least one entry in the clause is True (in other words, we should write $\vee$s between the elements of a clause, but do not to save space), and a formula consists of a set of clauses. The formula evaluates to True if all its clauses do. (We should write $\wedge$s between the clauses.) So the above example formula can be written $(\overline{x_1}, x_2)(x_3, x_2)$. It is easy to see that any Boolean formula can be written in this form, and it is easier to work with (in reductions, for example).

Theorem 9 (Cook-Levin Theorem)**.**
SATISFIABILITY is NP-complete.

Proof. It is easy to check that SATISFIABILITY is in P. A verifier, given a certificate consisting of a True/False assignment for the n variables, can simply evaluate the given Boolean formula.

Now we show that for any fixed language $\mathcal{L}$ in NP, there is a reduction $\mathcal{L} \leq_P$ SATISFIABILITY. By definition, since $\mathcal{L}$ is in NP, there exists a verifying TM $\mathcal{M}$ and polynomials p_1 and p_2 which bound respectively the length of a certificate and the time required to verify.

Given x (which may or may not be in $\mathcal{L}$), consider the running of $\mathcal{M}$ on input 'x, y' for some $y \in \{0,1\}^*$ of length at most $p_1(|x|)$ (which need not be a certificate). Let us consider that 'x, y' is represented as $x, \#, y$ on the tape. After time $\tau := p_2(|x| + p_1(|x|))$, if $\mathcal{M}$ has still not halted, we know that y is not a certificate for $x \in \mathcal{L}$, so running $\mathcal{M}$ further is not interesting. In this amount of time, $\mathcal{M}$ can read or write only the tape squares in the interval $[-\tau, \tau]$.

We will try to design a Boolean formula which is satisfiable if and only if there exists y with $|y| \leq p_1(|x|)$ which certifies $x \in \mathcal{L}$. The idea is the following: we will give a Boolean formula which simulates the running of $\mathcal{M}$ on inputs 'x, y' for any y with $|y| \leq p_1(|x|)$ for time τ, and returns True if and only if for some such y the machine $\mathcal{M}$ halts in the accepting state. If we can do this for each x, by definition we have shown $\mathcal{L} \leq_P$ SATISFIABILITY. It is convenient to insist that if $\mathcal{M}$ halts, then it does not in that time step or at any later time change the tape head position, state, or symbols on the tape.

We will do this construction in several steps (and we will not try to be 'economical'). First, we explain how to construct 'a tape square'. Take three variables b, o, z (blank, one, zero) and the clauses (b, o, z), $(\overline{b}, \overline{o})$, $(\overline{b}, \overline{z})$, $(\overline{o}, \overline{z})$. Observe that any satisfying assignment to b, o, z has exactly one of b, o, z set True; this corresponds to the point that a tape square has exactly one of the three possible symbols written on it. In our final formula, we will have $(2\tau+1)(\tau+1)$ tape squares, consisting of that many copies of this construction, using variables $b_{s,t}, o_{s,t}, z_{s,t}$ for $s \in [-\tau, \tau]$ and $t \in \{0, \ldots, \tau\}$; we think of this as representing the part of the tape of TM we are interested in (namely $[-\tau, \tau]$) at each time $0, 1, \ldots, \tau$.

Similarly, for each time $t = 0, \ldots, \tau$, we construct 'a state' for $\mathcal{M}$, consisting of $|Q|$ variables $q_{1,t}, \ldots, q_{|Q|,t}$ exactly one of which is true in any satisfying assignment, and 'a tape head position' consisting of $2\tau+1$ variables $p_{s,t}$ for $s \in [-\tau, \tau]$ exactly one of which is true in any satisfying assignment. We need $\tau + 1$ of each such construction.

This are all the variables we will need. We'll need more clauses, but let's repeat what we have so far. We can interpret any satisfying assignment of these variables as consisting of the contents of the tape interval $[-\tau, \tau]$, together with a tape head position and a state of $\mathcal{M}$, at each time $t = 0, \ldots, \tau$. What is missing is that we have not done anything to specify what the tape at $t = 0$ actually looks like (the input), or that the tape contents, state and tape head position should change over time according to the transition function of $\mathcal{M}$, or that we want to look only for simulations of $\mathcal{M}$ which halt in the accepting state.

The easy part is the input. For each $-\tau \leq s \leq -1$ and for $s = |x|$ we add the clause $(b_{s,0})$, for each $0 \leq s \leq |x| - 1$ we add $(o_{s,0})$ or $(z_{s,0})$ according to the sth letter of x, and for each $|x| + 1 \leq s \leq \tau - 1$ we add the clause $(\overline{b}_{s,0}, b_{s+1,0})$. Now in any satisfying assignment, the tape at time 0 contains 'x, y' written from the zero square for some $y \in \{0,1\}^*$, with all other squares blank.

Now we begin to specify that the simulation must actually follow the rules of $\mathcal{M}$. For each position $s \in [-\tau, \tau]$ and each time $0 \leq t \leq \tau - 1$, we specify that if the tape head is not at position s at time t, then the symbol at that position is the same at times t and $t+1$. In other words, we add the clauses $(p_{s,t}, \overline{b}_{s,t}, b_{s,t+1})(p_{s,t}, \overline{o}_{s,t}, o_{s,t+1})(p_{s,t}, \overline{z}_{s,t}, z_{s,t+1})$. Next, we consider what we do if the tape head is at position s. For each s and t, we need to change the symbol at position s according to the transition function of $\mathcal{M}$; so we add the clause $(\overline{p}_{s,t}, \overline{e}_{s,t}, \overline{q}_{i,t}, f)$ for each $e \in \{b, o, z\}$, each $1 \leq i \leq |Q|$, and with f chosen from $b_{s,t+1}$, $o_{s,t+1}$, $z_{s,t+1}$ according to the 'newsymbol' output by the transition function for input symbol in $\{\#, 0, 1\}$ corresponding to e and state i. We similarly need to change the state and the tape head position, which we do by adding the same clause with f chosen according to the transition function from the variables $q_{i,t+1}$, and with f chosen from $p_{s-1,t+1}$, $p_{s+1,t+1}$, $p_{s,t+1}$ according to whether the movement specified by the transition function is Left, Right or Halt; in the last case, by assumption, the symbol on the tape and the state do not change, so that they will all remain the same in all future time steps.

At this point, any satisfying assignment to this formula precisely corresponds to what $\mathcal{M}$ does for some input 'x, y', where $y \in \{0,1\}^*$ has length at most $\tau - x \geq p_1(|x|)$, up to time τ. We add the final clause $(q_{\tau,Y})$ where Y is the (number of the) accepting state to obtain a Boolean formula $F_{\mathcal{M},x,\tau}$. Any satisfying assignment of $F_{\mathcal{M},x,\tau}$ now gives an input 'x, y' on which $\mathcal{M}$ halts, in time at most τ, in the accepting state. Conversely, if there exists some y with $|y| \leq p_1(|x|)$ such that $\mathcal{M}$ halts in the accepting state in time at most τ for input 'x, y', then its behaviour up to time τ gives a satisfying assignment for $F_{\mathcal{M},x,\tau}$. Thus, by definition of $\mathcal{M}$, we have $x \in \mathcal{L}$ if and only if $F_{\mathcal{M},x,\tau}$ has a satisfying assignment.

It remains only to check that this construction of $F_{\mathcal{M},x,\tau}$ can be carried out in polynomial time from $\mathcal{M}$ and x. Since $\mathcal{M}$ is fixed independent of $|x|$, the total number of clauses is $O(\tau^2)$, and τ is a polynomial in $|x|$; so this construction is easily turned into an algorithm which finds $F_{\mathcal{M},x,\tau}$ in polynomial time in $|x|$, as desired. □

Now, the observation of Property 7 means that we can prove that a certain decision problem $\mathcal{L}$ in NP is NP-complete by showing that SATISFIABILITY is polynomial-time reducible to $\mathcal{L}$. And the more NP-complete problems we know, the more possible problems $\mathcal{L}_1$ we have to use Property 7 to show that a new problem $\mathcal{L}_2$ is NP-complete.

In fact, researchers nowadays are less and less interested in results that show a certain problem is NP-complete: there is a vast array of NP-complete problems, and knowing one more adds little to our understanding of complexity theory. (Although it does tell us something important about the problem.) It is more interesting to show that problems are in P, if only because there may be practical implications. Other topics of active research include learning about hard instances of decision problems that are NP-complete, and showing the relationships between other complexity classes.

The following graph theoretical problems are all known to be NP-complete:

HAM-CYCLE
Input: A graph G.
Question: Is there a Hamilton cycle in G?

CLIQUE
Input: A graph $G = (V, E)$ and an integer $K \geq 1$.
Question: Does G contain a clique of size K?

GRAPH-K-COLOURING (for any $K \geq 3$)
Input: A graph G.
Question: Does there exist a vertex colouring of G with K colours?

Note that GRAPH-2-COLOURING is in P, since checking if a graph is 2-colourable is the same as checking if it is bipartite, which can easily be done in polynomial time.

Given a graph $G = (V, E)$, a *stable set of G* (also called an *independent set*) is a set of vertices $S \subseteq V$ such that no two vertices in S are joined by an edge.

We can consider the following decision problems on finding stable sets in graphs.

STABLE-SET-OF-SIZE-K
Input: A graph G.
Question: Does G contain a stable set of size K?

STABLE-SET
Input: A graph G and an integer $K \geq 1$.
Question: Does G contain a stable set of size K?

HALF-STABLE-SET
Input: A graph $G = (V, E)$ with an even number of vertices.
Question: Does G contain a stable set of size $\frac{1}{2}|V|$?

Proposition 10.
For all $K \geq 1$, the problem STABLE-SET-OF-SIZE-K is in P.

Both STABLE-SET and HALF-STABLE-SET are NP-complete.

Proof. For STABLE-SET-OF-SIZE-K, note that we assume K is some fixed number in this case. To check if G contains a stable set of size K, we can just do a brute-force search of all subsets of K vertices, and check if one of them is stable. There are $\binom{|V|}{K} \leq |V|^K$ subsets of V with K vertices. It's not so hard to see that checking if a given subset $U \subseteq V$ is a stable set can be done in a polynomial number of steps. Hence checking if any of the subsets of V with K vertices is a stable set can also be done in a polynomial number of steps.

That STABLE-SET is NP-complete follows easily since CLIQUE is NP-complete, and a stable set in a graph G is exactly a clique in the complement $\overline{G}$.

HALF-STABLE-SET is Exercise 8. $\square$

3.9 Space Complexity

Until now we have only looked at languages/decision problems in terms of time complexity. That is, we were only looking at how long it took to determine the outcome. In this section we will have a short look at space complexity: how much 'memory' is required to solve certain problems?

A *2-tape Turing Machine* (deterministic by default) is similar to a normal deterministic Turing Machine, but has the following extra elements:

- Instead of a single tape, we have two tapes: the *input tape* and the *work tape*.
- The machine has also two *tape heads*, one for each tape. The tape head for the input tape can only read what is on that tape; the tape head for the work tape can both read and write on the work tape.
- The *transition function* is somewhat more involved, since it depends on both the contents of the square on the input tape and the square on the work tape. And it also controls the tape movements for both of the tape heads, which are independent. But it still only has to provide one write operation, since only the head for the work tape can write something on its tape.

So formally, the transition function is a function

$$\delta : (Q \setminus \{q_Y, q_N\}) \times \Sigma \times \Sigma \to Q \times \Sigma \times \{\text{Left}, \text{Right}, \text{Halt}\} \times \{\text{Left}, \text{Right}\},$$

where $\Sigma = \{0, 1, \#\}$. Note that there is no need to put 'Halt' in both movement lists; the machine simply stops.

When we want to consider space-limited Turing Machines, we always assume that the input x is written initially on the input tape, and that the work tape is completely blank when the calculations start.

A *nondeterministic 2-tape Turing Machine* is similar to a deterministic 2-tape Turing Machine, with the exception again that instead of a transition function we have a *multivalued transition mapping*

$$\delta : (Q \setminus \{q_Y, q_N\}) \times \Sigma \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{\text{Left}, \text{Right}, \text{Halt}\} \times \{\text{Left}, \text{Right}\}).$$

2-Tape Turing Machines appear to be more powerful than single tape Turing Machines: It's obvious how to mimic the behaviour of a single tape Turing Machine using a separate input and output tape.

But it's also not too hard to show that if a calculation can be done on a 2-tape Turing Machine, then the same calculation can be done on a single tape Turing Machine, using at most a polynomial factor extra time steps. One way to do this is by dividing the single tape into blocks of four squares. The first squares of each block indicate the input tape, the second squares correspond to the work tape, the third squares are all blank with the exception of one 0 or 1 to indicate the position of the input tape head, and the fourth squares are used similarly to mark the position of the work tape head.

The idea behind having a separate input and work tape is that when we talk about the 'amount of memory used', we can now concentrate on what is really used during the operations of the Turing Machine, not the amount of tape that was needed to provide the input. This is especially important when we want to talk about machines which 'use an amount of memory' which is much *smaller* than the size of the input (in the next section).

If we have a deterministic or nondeterministic 2-tape Turing Machine with a given input x on its input tape, then by the *amount of memory space* used during the calculations we mean the number of tape squares on the work tape over which the tape head passes at some point. (Where we don't count multiple times if the tape head is above some square more than once.)

The class *PSPACE* is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ for which there exists a 2-tape Turing Machine $\mathcal{M}$ and a polynomial $p(n)$, so that $\mathcal{M}$ decides, for every $x \in \{0,1\}^*$, if $x \in \mathcal{L}$ using an amount of memory space at most $p(|x|)$.

The class *NPSPACE* is the set of all languages $\mathcal{L} \subseteq \{0,1\}^*$ for which there exists a nondeterministic 2-tape Turing Machine $\mathcal{N}$ and a polynomial $p(n)$, so that $\mathcal{N}$ decides for every $x \in \{0,1\}^*$ if $x \in \mathcal{L}$ using an amount of memory space at most $p(|x|)$.

Theorem 11.
We have NP $\subseteq$ PSPACE*,* coNP $\subseteq$ PSPACE *and* PSPACE $\subseteq$ NPSPACE*.*

Proof. The third inclusion follows immediately from the definition (similar to the inclusion P $\subseteq$ NP).

For NP $\subseteq$ PSPACE, notice that if $\mathcal{L}$ is a language in NP, then for every $x \in \{0,1\}^*$ we have that $x \in \mathcal{L}$ if and only if there is a polynomial length certificate. So if we have all the time in the world, then we can just generate every possible certificate up to the polynomial length bound, and check if it gives us a certificate for a given $x \in \{0,1\}^*$. If one of the possible certificates works, we know $x \in \mathcal{L}$, otherwise we have $x \notin \mathcal{L}$. Since we don't have to store all the possible certificates, only generate them one by one (which we can do sequentially), all this testing can easily be done in a polynomial amount of memory.

The fact that coNP $\subseteq$ PSPACE can be proved in exactly the same way. $\square$

It is unknown if PSPACE really is larger than NP and coNP or not.

Earlier we discussed that the relation between P an NP is one of the most important problems in mathematics. Surprisingly, for PSPACE and NPSPACE we know what the situation is.

Theorem 12 (Savitch, 1970)**.**
We have PSPACE = NPSPACE*.*

This theorem is a beautiful consequence of the space complexity result on the decision problem *ST*-CONNECTIVITY discussed in the next section.

(Exercise: Why can we *not* just repeat the proof that $NP \subset PSPACE$?)

3.10 *ST*-CONNECTIVITY and the proof of Savitch's Theorem

Consider the following decision problem:

ST-CONNECTIVITY
Input: A graph G and two vertices u and v.
Question: Is there a path from u to v in G?

Standard algorithms to decide *ST*-CONNECTIVITY usually form some kind of spanning tree starting at one of the vertices. These algorithms need to store potentially many vertices in memory, to keep track of those vertices that are already visited during the search and those that are not. Hence if the graph G has N vertices, then these algorithms might need enough memory to store $O(N)$ vertices. Since the most efficient way to represent N vertices is by giving them numbers from 1 to N, storing one vertex in memory may need $\log(N)$ space. Hence the space complexity of traditional algorithms to decide if there is a path would be $O(N\log(N))$.

And at first sight, it doesn't seem possible to reduce that space complexity considerable. But in fact, *ST*-CONNECTIVITY can be decided, on a deterministic 2-tape Turing Machine, using far less memory.

Theorem 13 (Savitch, 1970).
It is possible to decide ST-CONNECTIVITY on a deterministic 2-tape Turing Machine using at most $O(\log^2(N))$ memory space, for a graph on N vertices.

Proof. We need to show that there is a constant C so that if G is a graph on N vertices, and u and v are two vertices of G, then deciding if there is a path from u to v can be done using at most $C\log^2(N)$ memory. In the rest of the proof, we use the shorter 'write to tape' to mean 'write on the work tape'.

For two vertices y_1, y_2 and a non-negative integer k, let $S(y_1, y_2; k)$ denote the statement 'there is a path from y_1 to y_2 in G of length at most 2^k'. Hence the statement 'there is a path from u to v in G' is equivalent to the statement $S(u, v; \lceil\log(N)\rceil)$.

The crucial observation about these statements is the following:

Observation. *If $k \geq 1$, then $S(y_1, y_2; k)$ holds if and only if there is a vertex z so that $S(y_1, z; k-1)$ and $S(z, y_2; k-1)$ hold.*

We can assume that the vertices are indicated by the integers from 1 to N. This means that to write a triple $(y_1, y_2; k)$ on the tape, where y_1, y_2 are vertices, and $k \leq \lceil\log(N)\rceil$, can clearly be done in at most $C'\log(N)$ bits, where C' is some constant. (We need at most $\lceil\log(N)\rceil$ bits for each of the vertices, at most $\lceil\log\lceil\log(N)\rceil\rceil$ bits for the integer k, and maybe a few extra bits (and pieces) to separate the parts of the triple.)

We will next prove the following claim:

Claim 1. *For vertices y_1, y_2 of G and k a non-negative integer, the claim $S(y_1, y_2; k)$ can be decided in $(k+1)\cdot 2C'\log(N)$ amount of memory.*

Proof. We prove the claim by induction on k.

Our general procedure is to write the statement we are investigating on the tape, and then after it we work out whether or not the claim is correct.

For $k = 0$, the claim $S(y_1, y_2; 0)$ is equivalent to 'y_1 and y_2 are adjacent in G'. To check that, we only need to write the triple $(y_1, y_2; 0)$ on the tape, and then read the whole input tape to see if there is an edge between y_1 and y_2 or not. (We have to write y_1 and y_2 on the tape, so that we can 'remember' which vertices we are looking at.) So the amount of space needed is at most $C'\log(N)$.

Now take $k \geq 1$. By our earlier observation, it is enough to check if there is a vertex z so that $S(y_1, z; k-1)$ and $S(z, y_2; k-1)$ hold. We can do this as follows

(i) Write the triple $(y_1, y_2; k)$ on the tape.
Set a equal to 1.

(ii) Next write the triples $(y_1, a; k-1)$ and $(a, y_2; k-1)$ on the tape.

(iii) Check if $S(a, y_2; k-1)$ holds. By induction, this can be done using $k\cdot 2C'\log(N)$ amount of memory. During this calculation, apart from the bits required to decide $S(a, y_2; k-1)$, we also keep the triples $(y_1, y_2; k)$ and $(y_1, a; k-1)$ on the tape. Hence the total amount of memory used in this step is indeed at most $k\cdot 2C'\log(N) + 2C'\log(N) = (k+1)\cdot 2C'\log(N)$.
If $S(a, y_2; k-1)$ does hold, go to step (iv), otherwise continue with step (v).

(iv) Erase the triple $(a, y_2; k-1)$ from the tape (and everything else left from the previous computation), and start deciding if $S(y_1, a; k-1)$ holds. As in step (iii), we can argue that this needs at most $(k+1)\cdot 2C'\log(N)$ amount of memory.
If $S(y_1, a; k-1)$ holds, then we now know that both $S(a, y_2; k-1)$ and $S(y_1, a; k-1)$ are true. So we have found a vertex z (namely the vertex with number a) so that both $S(y_1, z; k-1)$ and $S(z, y_2; k-1)$ hold, and hence we have decided that $S(y_1, y_2; k)$ is true. We can stop.

If $S(y_1, a; k-1)$ does not hold and $a < N$, then increase a by one (and erase everything left from the computation from the tape) and go back to step (ii).

If $S(y_1, a; k-1)$ does not hold and $a = N$, then we can conclude that there is no vertex z so that both $S(y_1, z; k-1)$ and $S(z, y_2; k-1)$ hold, and hence we have decided that $S(y_1, y_2; k)$ is false. We can stop.

(iv) If $S(a, y_2; k-1)$ does not hold, there is no need to check whether or not $S(y_1, a; k-1)$ holds. So, if $a < N$, we increase a by one (and erase everything left from the computation from the tape) and go back to step (ii).

If $a = N$, then we can conclude that there is no vertex z so that both $S(y_1, z; k-1)$ and $S(z, y_2; k-1)$ hold, and hence we have decided that $S(y_1, y_2; k)$ is false. We can stop.

(Observe that in the algorithm above a is used as a variable , but its value doesn't need to be additionally 'saved' on the tape, since it is always clear where to recover a from the information on the tape.) □

Claim 1 gives the theorem: all we have to do is decide $S(u, v; \lceil \log(N) \rceil)$. □

The same argument works when G is a directed graph on N vertices, with the only change being that $S(y_1, y_2, 0)$ becomes the statement 'y_1 and y_2 form a directed edge $\overrightarrow{y_1 y_2}$'. Calling the directed problem DIRECTED-*ST*-CONNECTIVITY, we get:

Corollary 14 (Savitch, 1970)**.**
It is possible to decide DIRECTED-ST-CONNECTIVITY on a deterministic 2-tape Turing Machine using at most $O(\log^2(N))$ *memory space, for a directed graph on N vertices.*

We now have all the tools to prove Savitch's Theorem.

Proof of Theorem 12. It is obvious that PSPACE ⊆ NPSPACE, so all we need to show is that NPSPACE ⊆ PSPACE. For this, let $\mathcal{L}$ be a language in NPSPACE. Hence there exists a nondeterministic 2-tape Turing Machine $\mathcal{N}$ and a polynomial $p(n)$, so that $\mathcal{N}$ decides for every $x \in \{0,1\}^*$ if $x \in \mathcal{L}$ using an amount of memory space at most $p(|x|)$.

Now we will describe the working of $\mathcal{N}$ in a slightly different way. By a *configuration* we will mean a complete description of the situation $\mathcal{N}$ can be in for a given input x. That means we must know what state the Turing Machine is in, what the position of the input tape head is, what the position of the work tape head is, and what the contents of the work tape are. We don't need to say what the contents of the input tape is, since that doesn't change.

There are $|Q|$ states (which is a fixed number). The input tape head can be in one of $|x|$ positions, while the work tape head can be in one of $p(|x|)$ positions. Finally, if we use an alphabet with a symbols (where the blank is one of the symbols), then there are at most $a^{p(|x|)}$ different relevant contents of the work tape, where this bound comes from the polynomial space usage assumption. So in total there are at most $|Q| \cdot |x| \cdot p(|x|) \cdot a^{p(|x|)}$ different configurations of $\mathcal{N}$ for a given input x. Let $C(x)$ be the collection of all those configurations, together with an extra 'configuration' Y.

Let $c_x \in C(x)$ be the configuration describing the initial state of $\mathcal{N}$ when given input x, i.e. when the state is q_0, the tape heads are at their first position, and the work tape consists of blanks only.

Next, we can interpret an allowed step of the nondeterministic Turing Machine as a directed edge from one configuration to the next one. And, given two configurations and a description (in particular, the multivalued transition mapping) of $\mathcal{N}$, it is easy to check if there is an allowed step from the first to the second configuration: we check if nothing has changed, apart from the changes prescribed by the transition mapping. So in this way we can see a calculation of $\mathcal{N}$ as a directed path in the directed graph which has as vertices all possible configurations. Finally, if a vertex in this directed graph corresponds to halting in state q_Y, we put an arc to Y.

Now $\mathcal{N}$ accepts an input x if and only if there is a finite sequence of allowed steps so that $\mathcal{N}$ halts in state q_Y. If there is such a finite sequence, then there definitely is such a finite sequence in which no configuration is repeated. In other words, $\mathcal{N}$ accepts an input x if and only if there is a path from c_x to Y, in the directed graph with vertex set $C(x)$ and directed edges as above.

By the arguments above and Corollary 14, it follows that there is a deterministic 2-tape Turing Machine that can decide this problem using at most $O(\log^2(|C(x)|))$ memory space. Since $|C(x)| \leq |Q| \cdot |x| \cdot p(|x|) \cdot a^{p(|x|)}$, we find that

$$\log(|C(x)|) \leq \log(|Q|) + \log(|x|) + \log(p(|x|)) + p(|x|)\log(a) = O(p(|x|)).$$

(Recall that $|Q|$ and a are just constants.) So the deterministic machine needs at most $O(p(|x|)^2)$ memory space. But since $p(|x|)^2$ is also a polynomial, the deterministic 2-tape Turing Machine can also decide whether or not $x \in \mathcal{L}$ in polynomial space.

So we can conclude that $\mathcal{L} \in$ PSPACE, completing the proof of NPSPACE $\subseteq$ PSPACE. □

It is tempting to think that Savitch's result that *ST*-CONNECTIVITY is solvable in $O(\log^2 N)$ memory space is optimal, but this turns out not to be true. Reingold (2004) showed that it can be done in $O(\log N)$ space: but the algorithm and proof of correctness are hard. It is an open problem to give a corresponding $O(\log N)$-space algorithm for DIRECTED-*ST*-CONNECTIVITY, or prove that no such algorithm exists.

3.11 Why proving P $\neq$ NP is hard

Suppose we want to prove P $\neq$ NP. (Even though it's probably not the easiest way to get a million dollars.) All we have to do is show that SATISFIABILITY (or any other language in NP) cannot be decided in polynomial time. But that means we have to look at all Turing Machines. Most Turing Machines will 'obviously' not solve SATISFIABILITY. Some will 'obviously' not halt in polynomial time. But it is quite possible that a Turing Machine solves SATISFIABILITY, yet the proof that it does so is very difficult. It is also quite possible that a Turing Machine halts in polynomial time, yet the proof that it does so is very difficult. There are examples of Turing Machines which solve number-theoretic problems which are known to halt in polynomial time if and only if (some version of) the Riemann Hypothesis holds (for example, the Miller primality test).

So we do not want to look 'too closely' at what Turing Machines are actually doing to solve a problem. Maybe we could try to use diagonalisation (as we did to see that not all languages are decidable) to prove P $\neq$ NP? Or maybe we can follow Turing, and show that if there is a machine which decides SATISFIABILITY in polynomial time, we can somehow modify it to make a machine whose existence is self-contradictory? (In fact, this is really diagonalisation in disguise.)

But this is not likely to work. Here is why.

An extension of the notion of a Turing Machine is a Turing Machine with oracle access. This is a Turing Machine which has an extra tape, the 'oracle tape', which starts blank and which may be read from and written to freely. The Turing Machine has two extra states, 'ASK' and 'RESPONSE'. If the Turing Machine enters the ASK state, then the contents of the oracle tape are changed and the Turing Machine moves to the RESPONSE state, from whence it continues operations normally.

The 'change' made is that if the oracle tape contains the word x, then it is replaced with the word $f(x)$, where f is 'the oracle'. It can be any function from words to words.

For example, suppose we choose a way of encoding pairs $(machine, input)$ as strings x.

We can let f be the function which maps the string x to Λ if x does not represent a pair $(machine, input)$, while if x does represent $(\mathcal{M}, y)$, then it returns 0 if $\mathcal{M}$ does not halt on input y, and 1 if $\mathcal{M}$ does halt on input y.

This function is not a computable function (because HALTING is not decidable) so this 'oracle' is not a computer. But this is a perfectly good theoretical definition. Now a Turing Machine with access to this function f, called the 'Halting Oracle' is basically the same as a normal Turing Machine, except that from time to time it can ask for an answer to an instance of HALTING.

It's easy to check that the proof from the previous section works just as well for Turing Machines with access to the Halting Oracle as for normal Turing Machines.

Exercise: Think about why the following logic is wrong: Running Turing's proof on Turing Machines with access to the Halting Oracle tells us that there is no Turing Machine with access to the Halting Oracle which can decide the Halting Problem. But the Halting Oracle decides the Halting Problem by definition. So we found a logical contradiction in mathematics.

Now suppose we had a proof that $P \neq NP$, and this proof somehow did not look too much at the inner workings of a Turing Machine. This proof would probably work just as well for Turing Machines with access to any fixed oracle.

To get the definitions of P and NP *relative to an oracle* f, written, a little confusingly, P^f and NP^f, just add 'with access to f' after each occurrence of 'Turing Machine' in the definitions of P and NP.

Theorem 15 (Baker, Gill, Solovay, 1975).
There is an oracle f such that P relative to f is equal to NP relative to f.

Proof sketch. The oracle f we use expects as input a triple $(\mathcal{M}, x, s)$ where $\mathcal{M}$ describes a Turing Machine, x is a word, and s is a number in unary. If it receives input in any other form, it returns the empty word Λ. It runs the machine $\mathcal{M}$ on input x until either the machine halts or more than s space is used. If $\mathcal{M}$ doesn't halt before using more than s space, then f returns Λ. If $\mathcal{M}$ halts in the accepting state, f returns 1, and if M halts in the rejecting state, then f returns 0.

This oracle can actually be computed by a (normal) Turing Machine F. What is more, the Turing Machine F runs in an amount of space at most polynomial in its input. (Basically because we can simulate Turing Machines efficiently.)

So if we have any language $\mathcal{L}$ which is in P relative to f, then there is a machine $\mathcal{M}$ with access to f which runs in polynomial time and decides $\mathcal{L}$. It's easy to check that replacing the oracle f with the machine F, we have a Turing Machine which in polynomial space decides $\mathcal{L}$. In other words, P relative to f is contained in PSPACE. In the other direction, if we have a language $\mathcal{L}_0$ in PSPACE, then by definition there is a Turing Machine $\mathcal{M}$ which decides it in polynomial space, where p is the polynomial. We can compute $p(|x|)$ in time polynomial in $|x|$, and so we can easily give a polynomial time Turing Machine with access to f which computes $s = p(|x|)$ and then asks the oracle f whether $\mathcal{M}$ accepts x in space s.

We see that P relative to f is PSPACE. So NP relative to f is certainly a superset of PSPACE. But we can repeat the construction: if $\mathcal{L}$ is in NP relative to f, then there is a nondeterministic machine $\mathcal{N}$ with access to f which accepts $\mathcal{L}$ in polynomial time. We can replace f with the machine F to find a nondeterministic machine which accepts $\mathcal{L}$ in polynomial space, so $\mathcal{L}$ is in NPSPACE. And since PSPACE = NPSPACE by Theorem 12, we are done. $\square$

Maybe this tells us that really P = NP? No, because the same authors also proved:

Theorem 16 (Baker, Gill, Solovay, 1975).
There is an oracle g such that P relative to g is not equal to NP relative to g.

Proof sketch. We can use the following oracle g. For each $n \geq 1$, at random, we choose a word w_n from $\{0,1\}$ of length n. Now g will return the first digit of w_n if $|x| = n$ and $x = w_n$, and otherwise g will return #. The language $\mathcal{L}$ we are trying to decide is the language of unary strings with n digits where the first digit of w_n is one.

Now a nondeterministic machine can guess $w_{|x|}$ in time $|x|$, given input x, and use g to accept $\mathcal{L}$ in polynomial time. But a deterministic machine cannot make use of g; whatever polynomial we choose, eventually a deterministic machine running for polynomial time doesn't have time to check the exponentially many possible words of length $|x|$ and will probably only get # out of g. So it has no hope of deciding $\mathcal{L}$. (This statement about probability takes some justification, and the justification is the Borel-Cantelli Lemma.) $\square$

So any proof that P $\neq$ NP (or the opposite!) has to make use of the fact that we are working with Turing Machines which do not have access to any oracle. (The jargon is: they do not relativise.) Your million-dollar proof will have to get its hands dirty and look into the details of what Turing Machines are really doing. If you want to prove P=NP this is easy: just give an algorithm. But to get P $\neq$ NP should now look hard.

Exercises

Exercise 1.
Let $f : \mathbb{N} \to \mathbb{R}_+$ be a function, and d a positive integer. Prove the following statement:

There exists a polynomial $p(n)$ of degree d so that $f(n) \leq p(n)$ for all $n \geq 1$, if and only if $f(n) = O(n^d)$.

Exercise 2.
In this question you are asked to design a Turing Machine for which the output is important. I.e. it's not important in what state the machine stops (so that can always be the positive state q_Y), but it is important what is left on the tape once the machine halts. Also, we always assume that the input x is a word from $\{0,1\}^*$. (I.e. the input contains no blank symbols.)

Here are two ways to let a Turing Machine make a 'copy' of a certain input $x \in \{0,1\}^*$:

(A) Upon input of a word $x \in \{0,1\}^*$ with its first (i.e. leftmost) symbol in the starting square of the input tape, the output of the Turing Machine must be two copies of x (and nothing else). One of the two copies still has its first symbol in the starting square, while the second copy is written to the left of the starting square, with a blank square in between to separate the copies.

Example: if the word is $x = 1011$, then the original input on the tape would have been $\cdots \# | \# | \underline{1} | 0 | 1 | 1 | \# | \# \cdots$ (The $| \cdot |$ indicate the squares; the underlined square is the starting square.) And the output must be $\cdots \# | \# | 1 | 0 | 1 | 1 | \# | \underline{1} | 0 | 1 | 1 | \# | \# \cdots$.

(B) Upon input of a word $x \in \{0,1\}^*$ with its first symbol in the starting square of the input tape, the output of the Turing Machine is a string twice as long as x, still with its first symbol in the starting square, and obtained by replacing each symbol of x by two copies of that symbol (and there's nothing else but blanks on the rest of the tape).

Example: if the word is $x = 1011$, the input is again $\cdots \# | \# | \underline{1} | 0 | 1 | 1 | \# | \# \cdots$. Then the output must be $\cdots \# | \# | \underline{1} | 1 | 0 | 0 | 1 | 1 | 1 | 1 | \# | \# \cdots$.

Choose one of two ways above to make a copy of a string x, and design a Turing Machine that performs the task of making such a copy. Make sure that you not only describe the formal details, but also explain the ideas behind your approach.

Exercise 3.
Determine, justifying your answers, if the following statements are true for all languages $L_1, L_2 \subseteq \{0,1\}^*$.

(a) *If* L_1 *and* L_2 *are in NP, then* $L_1 \cup L_2$ *is in NP.*

(b) *If* L_1 *and* L_2 *are in NP, then* $L_1 \cap L_2$ *is in NP.*

(c) *If* L_1 *and* L_2 *are in NP, then* $L_1 \setminus L_2$ *is in NP.*

Exercise 4.
Show that $\leq_P$ *is a quasi-order on the set of languages.*

Let $\mathcal{L}$ *be a language in P; show that* $\{\mathcal{L}\}$ *is an antichain of* $\leq_P$*-minimal elements.*

Exercise 5.
Sketch a proof of Property 7.

Exercise 6.
Give a sketch of a proof of Property 8. I.e. prove that if $\mathcal{L} \in \mathrm{P}$*, then for every other language* $\mathcal{L}' \in \mathrm{P}$ *we have that* $\mathcal{L}' \leq_P \mathcal{L}$*.*

Exercise 7.
Show that the problem 3*-SAT is NP-complete, where* 3*-SAT is the special case of SATISFIABILITY where we insist that the input Boolean formula must be in conjunctive normal form and all clauses have size exactly three.*

Describe informally an algorithm which decides 2*-SAT (defined analogously to* 3*-SAT) in polynomial time; explain why the running time is guaranteed to be polynomial.*

Exercise 8.
Show that HALF-STABLE-SET is NP-complete.

Exercise 9.
Decide if the following decision problem can be decided in $O(\log^2(N))$ *amount of memory, for a graph on* N *vertices:*

CYCLE-THROUGH-VERTEX
Input: *A graph* G *and one vertex* u*.*
Question: *Is there a cycle in* G *through* u*?*