LTCC Course on Graph Theory 2013/14

Notes 3

Complexity and Algorithms

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

The default textbook and source for anything algorithmic is: T.H. Cormen, C.E. Leiverson, R.L. Rivest and C. Stein, *Introduction to Algorithms* (2nd edition), MIT Press, 2001 (ISBN: 0262531968).

An alternative (and lighter in weight) source is H.S. Wilf, *Algorithms and Complexity* (2nd edition), Peters, 2003 (ISBN: 1568811780). The first edition (which is more than enough for us) can be downloaded for free from `www.math.upenn.edu/~wilf/AlgoComp.pdf`.

The default book for complexity theory is still, after 30 years (!), and still in print, M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, 1979 (ISBN: 0716710455). The only serious alternative I can think of is C.H. Papadimitriou, *Computational Complexity*, Addison Wesley, 1994 (ISBN: 0201530821).

3.1 Order of Functions

For functions $f, g : \mathbb{N} \longrightarrow \mathbb{R}$, we define the following order comparisons:

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

* $f(n) = o(g(n))$, pronounced "*$f(n)$ is small oh $g(n)$*", or "*...little 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 $|f(n)| \leq c\,|g(n)|$.

3.2 Decision 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 $b = -6969696969$".

- From now on we will use the word *machine* to indicate some kind of (real or abstract) device that is meant to "solve" one particular problem. Examples of machines you can think of are a computer, a computer program, an algorithm, but also a mechanical cash register.

  Note that the above means that a machine is only supposed to be able to deal with exactly one problem : but with every possible instance of that particular problem. So a machine doesn't have to be something that is "programmable". It must have some way to deal with input, and a way to give its output.

- 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 outcome 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 making a 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 $x, y, z \geq 1$ integers ?

  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!

3.3 Words and Languages

- * An *alphabet* Σ is a finite set of symbols.

We will often, but not always, use the alphabet $\Sigma = \{0, 1\}$.

* Given an alphabet Σ, 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 $\{2, x\}^* = \{\,\Lambda, 2, x, 22, 2x, x2, xx, 222, \dots\}$.

* 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—and any quick algorithm would leave the worlds 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.

3.4 Turing Machines

A *Turing Machine* (more precisely, a *deterministic* Turing Machine) has the following elements:

* A finite alphabet Σ. This alphabet is supposed to have one special symbol #, which stands for the blank symbol.
  Unless otherwise specified, we will assume $\Sigma = \{0, 1, \#\}$.
* A 2-way infinite *tape*, divided into *squares*. There is one special square, the *starting square*. Each square contains one symbol from the alphabet Σ; and only a finite number of squares contain a symbol that is not the blank symbol.
* A *tape head*, that can read the contents of one square at a time, that can write a symbol from the alphabet on that square, and that can move to the left or right along the tape, but only one square at the time.
* A finite lists of states $Q = \{q_0, q_Y, q_N\} \cup \{q_1, \ldots, q_s\}$. Here q_0 is the initial state, q_Y is the final state in which the answer should be "Yes", and q_N is the final state in which the answer should be "No".
* A program in the form of a *transition function*

  $$\delta : (Q \setminus \{q_Y, q_N\}) \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, +1\},$$

  which should be read as

  $$\delta : (\textit{state}, \textit{symbol}) \quad \longrightarrow \quad (\textit{newstate}, \textit{newsymbol}, \textit{head movement}).$$

  Here *state* is the current state of the machine, *symbol* is the symbol currently written on the square the tape head can read, *newstate* is the new state of the machine, *newsymbol* is the symbol the tape head will write on the square below it, and *head movement* is an indication of the head movement ("-1" one square to the left, "0" no movement, and "$+1$" one square to the right).

  Note that the tape head is assumed to write the new symbol before it makes a move.

Also note that the tape head always writes a new symbol (and hence overwrites the old symbol). If you want to mimic the situation that the old symbol is left in place, you can take *newsymbol* the same as *symbol*.

- A Turing Machine is supposed to be used as follows:
  - The input x (where x is a word from Σ^*) is given to the machine by writing it on the tape. By default, the first symbol of x is written on the starting square, and then the rest of the symbols on the squares to the right. All other squares contain the blank symbol.
  - The machine starts in the starting state q_0, with the tape head above the starting square. From then on it performs one action each time step, according to the transition function.
  - The machine stops immediately when it reaches one of the final states q_Y or q_N.
- * A word $x \in \Sigma^*$ is *accepted* by a Turing Machine $\mathcal{M}$ if, when given x as input, $\mathcal{M}$ halts after a finite number of steps in state q_Y.
  - * A word $x \in \Sigma^*$ is *rejected* by a Turing Machine $\mathcal{M}$ if, when given x as input, $\mathcal{M}$ halts after a finite number of steps in state q_N.
  - * A Turing Machine $\mathcal{M}$ *accepts a language* $\mathcal{L} \subseteq \Sigma^*$ if for all $x \in \Sigma^*$ we have that $\mathcal{M}$ accepts x if and only if $x \in \mathcal{L}$.

  Note that if $\mathcal{M}$ accepts a language $\mathcal{L}$, then there is an asymmetry between words in $\mathcal{L}$ and words not in $\mathcal{L}$: If $x \in \mathcal{L}$, then $\mathcal{M}$ halts after a finite number of steps in the "Yes"-state. But if $x \notin \mathcal{L}$, then there are two possible outcomes: either $\mathcal{M}$ halts after a finite number of steps in state q_N, or $\mathcal{M}$ never halts.

  - * A Turing Machine $\mathcal{M}$ *decides a language* $\mathcal{L} \subseteq \Sigma^*$ if for all $x \in \Sigma^*$ we have that if $x \in \mathcal{L}$, then $\mathcal{M}$ accepts x, while if $x \notin \mathcal{L}$, then $\mathcal{M}$ rejects x.

  As indicated earlier, we will identify the phrases "$\mathcal{M}$ solves the decision problem Π" and "$\mathcal{M}$ decides the corresponding language $\mathcal{L}_\Pi$".

- – As an example, take $\Sigma = \{0, 1, \#\}$ and $Q = \{q_0, q_Y, q_N\} \cup \{q_1, q_2, q_3, q_4, q_5\}$.
  - And the transition function δ : (*state*, *symbol*) $\longrightarrow$ (*newstate*, *newsymbol*, *head movement*) :

state	*symbol*	*newstate*	*newsymbol*	*movement*
q_0	0	q_1	#	$+1$
q_0	1	q_2	#	$+1$
q_0	#	q_Y	.	.
q_1	0	q_1	0	$+1$
q_1	1	q_1	1	$+1$
q_1	#	q_3	#	-1
q_2	0	q_2	0	$+1$
q_2	1	q_2	1	$+1$
q_2	#	q_4	#	-1
q_3	0	q_5	#	-1
q_3	1	q_N	.	.
q_3	#	q_Y	.	.
q_4	0	q_N	.	.
q_4	1	q_5	#	-1
q_4	#	q_Y	.	.
q_5	0	q_5	0	-1
q_5	1	q_5	1	-1
q_5	#	q_0	#	$+1$

- We often indicate the transitions by a flow diagram. Here is a simple example of a Turing Machine with $\Sigma = \{0, 1, \#\}$, $Q = \{q_0, q_Y, q_N\} \cup \{q_1\}$, and transition function:

state	*symbol*	*newstate*	*newsymbol*	*movement*
q_0	0	q_0	0	$+1$
q_0	1	q_1	1	$+1$
q_0	#	q_N	.	.
q_1	0	q_1	0	$+1$
q_1	1	q_N	.	.
q_1	#	q_Y	.	.

The flow diagram of this transition function would be:

$$\begin{tikzcd}
q_0 \arrow[loop above, "0:0,+1"] \arrow[rr, "1:1,+1"] \arrow[dr, "\#:.,."] & & q_1 \arrow[loop above, "0:0,+1"] \arrow[dl, "1:.,."'] \arrow[dr, "\#:.,."] & \\
& q_N & & q_Y
\end{tikzcd}$$

Here something like $\begin{tikzcd} q \arrow[r, "a:b,i"] & q' \end{tikzcd}$ means: when in state q and reading symbol a, change to state q', write symbol b and move the tape head by the increment i.

3.5 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. (Recall that $|x|$ is the length of x, i.e., the number of symbols in the word x.)

  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, $\mathcal{M}$ 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 1

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

  Proof We easily have P ⊆ 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 state q_Y after at most $p(|x|)$ steps, while for a word $x \notin \mathcal{L}$ the Turing Machine either halts in state q_N or doesn't halt at all. Now we design a new Turing Machine $\mathcal{M}^P$ 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, $\mathcal{M}^P$ 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 $\mathcal{M}^P$ decides the language $\mathcal{L}$.

  The one remaining issue is if $\mathcal{M}^P$ works in polynomial time. I.e., 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.

  The above shows that $\mathcal{M}^P$ 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. ■

- Because of the close relation between languages and decision problems, we also say about decision problems that they are in P.

3.6 Nondeterministic Turing Machines

A *nondeterministic Turing Machines* has all the elements of a normal Turing Machine, with one exception :

* Instead of a transition function, we have a *multivalued transition mapping*

$\delta' : (Q \setminus \{q_Y, q_N\}) \times \Sigma \longrightarrow$
P(Q×Σ $\times \{-1, 0, +1\}) \setminus \varnothing$.

(Here $\mathcal{P}(X)$ for a set X denotes the collection of subsets of X.)

And 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 what 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 "oracle" housed in the machine who decides which of the allowed steps to take, but we have no clue how this oracle comes to its decisions, if it always will make the same decisions in the same situation, etc.

- * 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 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 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 state q_N. 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 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.

- **Property 2**

  We have P ⊆ 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 1 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. ■

3.7 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
    - $|y| \le p_1(|x|)$;
    - $\mathcal{M}$ 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 3**

  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 built deterministic Turing Machine and use it as 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.) Once that is done, $\mathcal{N}$ simulates all the steps $\mathcal{M}$ would have done with the combined input "x, z" (hence 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. ■

3.8 Complements of Languages and Decision Problems

For humans, the following decision problems would be 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 have coP and coNP.

- Following our identification of languages and decision problems, we can also look as 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:

$$\mathcal{L}_\Pi^c = \{\, x \in \Sigma^* \mid x \text{ is not the proper encoding of a graph, or } x \text{ is the encoding of a graph that is not 3-colourable} \,\}.$$

But this is not really the natural way to think about it. So we usually just ignore the possibility that x is not the proper encoding of a 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

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

- **Property 4**

  We have the following relations between some of the classes above:

– *$P = coP$;*
– *$P \subseteq NP$ and $P \subseteq coNP$ (hence $P \subseteq (NP \cap coNP)$).*

Proof The fact that P = coP follows from the definition of P as language 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 P ⊆ NP follows from Property 2 and Theorem 3. ■

- It is unknown if P is equal to NP ∩ 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/`).

3.9 Polynomial Reducibility and Completeness

So far we have a lot of definitions and abstract classes, but we haven't seen much that one can actually *do* with Turing Machines. Nor will we see anything much explicitly (it takes too long). However the following is true.

Universal Computation: For any non-interactive program in Java (or C, or BASIC, or any other programming language) there is a Turing Machine whose outputs are identical. Furthermore, this Turing Machine takes longer by at most a polynomial amount, and uses at most a polynomial amount more memory. In particular, since it's easy to write efficient Turing Machine simulators in Java, there exist Turing Machines which can efficiently simulate any Turing Machine (given as input).

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 $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, 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* $\mathcal{L}_2$, notation $\mathcal{L}_1 \leq_P \mathcal{L}_2$, if there exists a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, 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 $\mathcal{L}' \in$ 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 5

– *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 P = NP.*

Property 6

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. (exercise)

Somewhat less obvious is the following.

Property 7

Every language in P is P-complete. (exercise)

- Since all 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. Since then, the observation of Property 6 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 6 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
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 $\frac{1}{2}|V|$?

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

Property 8
– *Both STABLE-SET and HALF-STABLE-SET are NP-complete.*
– *For all $K \geq 1$, The problem STABLE-SET-OF-SIZE-K is in P.*

Proof of 2nd part 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. ■

3.10 Space Complexity

Until now we have only looked at languages / decision problems in terms of time complexity. I.e., 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. 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
    $$\delta'' : (Q \setminus \{q_Y, q_N\}) \times \Sigma \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, +1\} \times \{-1, 0, +1\},$$
    where $\Sigma = \{0, 1, \#\}$.

We always assume that the input x for a 2-tape Turing Machine 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*

$$\delta''' : (Q \setminus \{q_Y, q_N\}) \times \Sigma \times \Sigma \longrightarrow \mathcal{P}(Q \times \Sigma \times \{-1, 0, +1\} \times \{-1, 0, +1\}).$$

- Note that 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.
  - 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 largest number of squares of the work tape with non-blank symbols on them at any time during the calculations.
  - 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 9**

  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, 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. ■

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

Theorem 10 (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.

3.11 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 11** (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 *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.*

Once the claim is proved, the theorem follows, since we've already seen that all we have to do is decide $S(u, v; \lceil \log(N) \rceil)$.

Proof of Claim 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. 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:

1. Write the triple $(y_1, y_2; k)$ on the tape.

   Set a equal to 1. (This a is used as a variable in the steps below, but doesn't need to be written on the tape, since every time we know exactly where on the work tape the number a can be found.)

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

3. 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 4a, otherwise continue with step 4b.

4a. Erase the triple $(a, y_2; k-1)$ from the tape, and start deciding if $S(y_1, a; k-1)$ holds. As in step 3, 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 go back to step 2.

   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.

4b. 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 replace the two triples $(y_1, a; k-1)$ and $(a, y_2; k-1)$ by $(y_1, a+1; k-1)$ and $(a+1, y_2; k-1)$ and continue from step 3.

   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.

The algorithm above proves the induction step and hence completes the proof of the claim and also of the theorem. ■

- The same arguments as above can prove that if G is a directed graph on N vertices, then the following decision problem can be decided in $O(\log^2(N))$ amount of memory.

  DIRECTED-VERTEX-SET-CONNECTIVITY
  Input: A directed graph G, a vertex u and a subset S of vertices.
  Question: Is there a directed path in G from u to a vertex in S ?

  (For each $v \in S$, deciding if there is a directed path from u to v can be done completely identical to the undirected case above (that proof is directed agnostic). So to decide if there is a path from u to some vertex of S, we just have to try one by one the vertices of v. As soon as we get one positive output, we are done. And when we move from one failed vertex to the next, we don't need to keep the failed vertices on tape.)

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

  Theorem 10 (Savitch, 1970)
  We have PSPACE = NPSPACE.

  Sketch of Proof It is obvious that PSPACE $\subseteq$ NPSPACE, so all we need to show is that NPSPACE $\subseteq$ 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 is. (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. (The work tape is supposed to be infinitely long in both directions, but since at most $p(|x|)$ squares of it will be used, we only have a finite number of different contents.) 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.

  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. (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.

Now let $S_Y \subseteq C(x)$ be the set of all possible configurations in which the state is q_Y. Then $\mathcal{N}''$ 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 a configuration in S_Y, in the directed graph with vertex set $C(x)$ and directed edges given by allowed steps.

By the arguments above and Theorem 11, 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 $|C(x)| \le |Q| \cdot |x| \cdot p(|x|) \cdot a^{p(|x|)}$, we find that

$$\log(|C(x)|) \;\le\; \log(|Q|) + \log(|x|) + \log(p(|x|)) + p(|x|)\,\log(a) \;=\; O(p(|x|))$$

(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.

3.12 The Halting Problem

If we had infinite time and memory, could we compute everything? To put it another way, is it true that for any language $\mathcal{L}$ there is a Turing Machine which decides $\mathcal{L}$ if we let it run long enough?

Exercise *No.*

So there is a language $\mathcal{L}$ that is *undecidable*. But diagonalisation doesn't give us an example, and the proof doesn't work for 'reasonable' languages which come from 'real problems'. Maybe every language we can describe finitely is decidable?

Theorem 12 (Turing, 1936) *The Halting Problem is undecidable.*

The Halting Problem is the following.

HALTING
Input: A Turing Machine M and a word x.
Question: Does M halt given input x?

This is actually very easy to prove.

Sketch of Proof: Suppose that HALTING is decidable. Then there is a machine H that decides it. The input to H is expected to be of the form (M, x), and M is represented by a string.

First we create a machine which takes an input y and duplicates it to (y, y). We 'bolt this on' to the front of H (meaning we run it then we start H). Then we change the state q_Y of H to a 'normal' state q_r with all three transitions returning to q_r. To be technically correct we create a 'new' state q_Y, but we don't have any transition which goes to it. We call this machine B.

Now suppose we feed x into B. Then the first part of B will (effectively) ask H to interpret x as a Turing Machine, and see whether the Turing Machine represented by x halts when given input x. And the modification we made to q_Y means that B will halt in q_N if the TM represented by x does *not* halt given input x, while it will run forever otherwise.

So what happens when we feed a string b representing the Turing Machine B into B? $\square$

Of course, HALTING is not a 'natural' problem. But Matiyasevich in his 1970 PhD thesis, building on work of Davis, Putnam and Robinson, proved that the language of Diophantine equations (polynomials in many variables with integer coefficients) which have integer solutions is undecidable—this is a central concern of Number Theory, and the result finally kills Hilbert's dream that any mathematical problem can be solved given sufficient effort.

3.13 Why proving P $\neq$ NP is hard

Suppose we want to prove P $\neq$ NP. 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.

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?

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 (M, y) then it returns 0 if M does not halt on input y, and 1 if 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 both proofs from the previous section work 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' just add 'with access to f' after each occurrence of 'Turing Machine' in the definitions of P and NP)

Theorem 13 (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 (M, x, s) where 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 Λ. It runs the machine M on input x until either the machine halts or more than s space is used. If M doesn't halt before using more than s space, then f returns Λ. If M halts in the accepting state, f returns 1, and if f halts in the rejecting state then f returns 0.

This oracle can actually be computed by a 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, there is a machine 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 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 $\mathcal{L}'$ in PSPACE then there is a machine M which decides it in polynomial time, 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 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 M 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 we are done. □

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

Theorem 14 (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 Λ 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). □

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*).

Exercises

1 Let $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ 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 $f(n) = O(n^d)$.

2 Decide the languages accepted or decided by the Turing Machines at the bottom of page 4 and at the top of page 5.

3 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 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 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.

4 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.

5 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.

6 Sketch a proof of Property 6.

7 Give a sketch of a proof of Property 7. I.e., prove that if $\mathcal{L} \in$ P, then for every other language $\mathcal{L}' \in$ P we have that $\mathcal{L}' \leq_P \mathcal{L}$.

8 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?