Complexity Theory

Supervision 3 – Solutions

6. NP, co-NP, and UP

1. It is often claimed that a proof of the proposition $\mathrm{P} = \mathrm{NP}$ would have drastic consequences: it would let us solve difficult optimisation problems efficiently, but would also break security and e-commerce by making public-key cryptography impossible. What objections could be made against such a claim?

   > A proof of $\mathrm{P} = \mathrm{NP}$ would certainly be a breakthrough in the study of computer science and mathematics and would be a great surprise to most of the research community. The question $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ is of course quite a thorny one with constant heated debate from both sides – but the fact remains that despite $\mathrm{P} = \mathrm{NP}$ being a problem that should be very "easy" to prove (just provide a single poly-time algorithm to an NP-complete problem), many decades of intense study has not managed to accomplish this. More interestingly, there are a lot of "close calls": algorithms which are just about exponential and would have slipped into P if only there was a slightly different detail or parameter in the proof. Sometimes these close calls get misinterpreted, which may be why the list of proposed attempts at deciding $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ is currently in favour of $\mathrm{P} = \mathrm{NP}$. But, as Scott Aronson put it, "*like any other successful scientific hypothesis, the* $\mathrm{P} \neq \mathrm{NP}$ *hypothesis has passed severe tests that it had no good reason to pass were it false*" – any other empirical, experiment- or measurement-based science would have concluded that $\mathrm{P} \neq \mathrm{NP}$ decades ago.
   >
   > Still, the question remains open, so there *is* a chance that P might equal NP. Does that lead to immediate societal collapse all of our security systems would be compromised? Or do we finally get around to building the flying cars we were promised? The truth is, even if $\mathrm{P} = \mathrm{NP}$, our lives may not be impacted significantly.
   >
   > The most obvious issue is that the proposed poly-time algorithm for an NP-complete problem may still be impractical – P is a very broad class and includes asymptotic complexities that a programmer would simply laugh at. While it is the case that most of our "standard" poly-time algorithms rarely exceed $O(n^4)$, the constants and exponents can stack up if we go through several reduction steps. For example, imagine we find a poly-time algorithm for a very niche, obscure NP-complete problem – generalised Pokémon, for example? In theory, this means we have conclusive proof that our cryptography systems are insecure, but we can also reap the benefits of tractable algorithms for protein folding. How would we go about actually getting there? We can either try to reduce protein folding to generalised Pokémon (problematic), or reduce it to SAT using Cook's theorem, then go through the chain of reductions that established that generalised Pokémon was NP-complete, and compose the algorithms. Yes, the algorithm will still be polynomial, but ridiculously inefficient. In addition, you don't even need the algorithm to be $O(n^{1000})$ to call it impractical – it may

well be $O(n)$, only with an enormous constant factor. It's easy to forget about constant factors: the time to render an animated film is linear in the number of frames, but if every frame takes a day to render, you're looking at either a very short film or a very long wait (or lots of computers, as in the case of *Toy Story* – but the 117-machine render farm still completed about three minutes of the film per week).

Another interesting issue is that the proof $\mathrm{P} = \mathrm{NP}$ may be *nonconstructive* – that is, it may not actually give a useful (even in the broad sense) algorithm for solving NP problems. Notably, this is the view held by Donald Knuth (Question 17), who knows a thing or two about computing. $\mathrm{P} = \mathrm{NP}$ can be interpreted as an existence proof, saying that there exists an algorithm that can solve some particular NP problem in polynomial time. However, there may not be a witness of this statement, i.e. we may not be able to extract the actual algorithm, despite knowing that somewhere it exists. Or, for example, imagine proving the statement "there do not exist NP-intermediate problems", but your proof proceeds by contradiction: if there exists an NP-intermediate problem, we get a logical contradiction. This establishes $\mathrm{P} = \mathrm{NP}$ (by Ladner's theorem), but brings us no closer to breaking RSA or revolutionising medicine. Such a possibility would certainly encourage further research into actually finding a poly-time algorithm for an NP-complete problem, but there is still no guarantee that it can be found in reasonable time.

Still another possibility is that the decision problem "is $\mathrm{P} = \mathrm{NP}$?" may itself be proved to be unprovable or undecidable (or more precisely, independent of the axioms of ZFC). This would also be an unusual result, but at this point even getting a result would be unusual.

2. The complexity class NP is closed under which of the following set-theoretic operations: intersection, union, complement? Briefly justify your answers.

Let L and K be two languages in NP which are verifiable in $O(n^l)$ and $O(n^k)$, respectively.

- The intersection of L and K is still in NP: the verifier for $L \cap K$ simply runs the verifier for the two languages one after the other and accepts only if both accept. This can be done in $O(n^{\max(k,l)})$, which is still polynomial.
- Similarly $L \cup K$ is in NP: run both verifiers and accept if at least one of them accepts.
- It is unknown if NP is closed under complement – this is exactly the question $\mathrm{NP} \stackrel{?}{=} \text{co-NP}$, still unresolved.

3. Prove or disprove the following claims, or show that it is an open problem:

a) If $L, K \in \text{co-NP}$ then $L \cup K \in \text{co-NP}$.

By definition, $L, K \in \text{co-NP}$ if $\overline{L}, \overline{K} \in \mathrm{NP}$. Now,

$$L \cup K \in \text{co-NP} \iff \overline{L \cup K} \in \mathrm{NP} \iff \overline{L} \cap \overline{K} \in \mathrm{NP}$$

and the last condition holds since NP is closed under intersection.

b) If $L \in \mathrm{NP}$, $K \subset L$ and $K \in \text{co-NP}$ then $L \setminus K \in \mathrm{NP}$.

> Follows from $L \in \text{NP}$, $\overline{K} \in \text{NP}$, $L \setminus K = L \cap \overline{K}$, and closure of NP under intersection.

c) If L is NP-complete, then $D = \{ xx \mid x \in L \}$ is NP-complete.

> The language D is in NP: given any input we can split it into two halves, check that the halves are equal, and check if a half is in L, all in polynomial time. To show that D is NP-hard, we reduce an NP-complete language to it – L itself is a good choice, with the reduction $f(x) = xx$. Note that, in general, NP-complete languages are not closed under concatenation (or any other standard operation on languages).

4. Show that a language L is in co-NP if, and only if, there is a nondeterministic Turing machine M and a polynomial p such that M halts in time $p(n)$ for all inputs x of length n, and L is exactly the set of strings x such that *all* computations of M on input x end in an accepting state.

> This question nicely captures the duality between NP and co-NP. Consider a nondeterministic Turing machine M which halts on all inputs in polynomial time. The main property of M that we use is that the language of strings for which *every* computation is accepting is the complement of
>
> ($\Rightarrow$) Assume L is in co-NP, that is, $\overline{L} \in \text{NP}$. By definition, there exists an NDTM $M_{\overline{L}}$ which accepts $\overline{L}$ time bounded by a polynomial p. We can construct a machine M_L as follows: on input x, simulate $M_{\overline{L}}$ for at most $p(|x|)$ steps, and end in state `acc` if $M_{\overline{L}}$ rejects or runs out of time, and `rej` otherwise. By construction, M_L runs for at most $p(|x|)$ steps for an input $x \in L$. To show that L is the set of strings on which *all* computations of M_L end in an accepting state, it is sufficient to show that the complement $\overline{L}$ is the set of strings for which there *exists* a computation of M_L that ends in a rejecting state. A string x is in $\overline{L}$ if and only if there exists an accepting computation of $M_{\overline{L}}$ on x, which, by construction, is equivalent to there existing a rejecting computation of M_L, as required.
>
> ($\Leftarrow$) Assume there exists a machine M_L and polynomial p such that M halts in time $p(|x|)$ for all inputs x and L is the set of strings for which all computations of M end in an accepting state. We need to prove that $L \in$ co-NP, or equivalently, $\overline{L} \in \text{NP}$. We define the suitable nondeterministic TM $M_{\overline{L}}$ that accepts $\overline{L}$ by swapping the rejecting and accepting states of M_L. $M_{\overline{L}}$ halts in polynomial time $p(|x|)$ since so does M_L. Next, we take an $x \in \overline{L}$ and prove that there exist an accepting computation of $M_{\overline{L}}$ on x. By construction, it is sufficient to show that there exists a rejecting computation of M_L on x; but we know that this must be the case because the set of strings for which this is *not* the case is exactly L, but x is an element of the complement $\overline{L}$.

5. Define a *strong* nondeterministic Turing machine as one where each computation has three possible outcomes: accept, reject or maybe. If M is such a machine, we say that it accepts L, if for every $x \in L$, every computation path of M on x ends in either accept or maybe, with at least one accept, *and* for $x \notin L$, every computation path of M on x ends in reject or maybe, with at least one reject.

   Show that if L is decided by a strong nondeterministic Turing machine running in polynomial

time, then $L \in \text{NP} \cap \text{co-NP}$.

> Assume L is decided by a strong NTDM M in time bounded by a polynomial p. We need to show that L is both in NP and co-NP by a simulation argument. Construct a (standard) NDTM M_1 that simulates M, but when M would halt in the maybe state, M_1 goes into an infinite loop. If M accepts L, there must be at least one accepting computation for every string $x \in L$ (and any number of maybes), so M_1 will accept (again by the standard definition of acceptance by an NDTM). Similarly, construct M_2 from M_1 by swapping accepts and rejects, so if M ends in at least one reject for an $x \in \overline{L}$ (and any number of maybes), M_2 will end in at least one accept for strings in L.

6. We saw in the lectures that if there is a one-way function, then there is a language L in UP that is not in P. Suppose that the RSA function described in the lecture notes (page 38) is a one-way function. What is the language L that can then be proved to be in $\text{UP} \setminus \text{P}$?

> Following the proof in the notes, fix positive integers e, p, q and define the language L as
>
> $$L \triangleq \{\, (b, y) \mid \exists x.\ x \leq b \ \wedge\ x^e \bmod pq = y \,\}$$
>
> Alternatively, if we do not take e, p, q to be fixed, suppose we have two injections $g : \mathbb{N}^3 \rightarrowtail \mathbb{N}$ and $h : \mathbb{N}^4 \rightarrowtail \mathbb{N}$, and define L as
>
> $$L \triangleq \{\, (b, y) \mid \exists x, e, p, q.\ h(x, e, p, q) \leq b \ \wedge\ g(x^e \bmod pq, pq, e) = y \,\}$$
>
> The injections ensure that the input 4-tuples and output triples are mapped to (unique) natural numbers that can be compared.

7. Space complexity

1. Show that, for every nondeterministic machine M which uses $O(\log n)$ work space, there is a machine R with three tapes (`input`, `work` and `output`) which works as follows: on input x, R produces on its output tape a description of the configuration graph for M, x, and R uses $O(\log |x|)$ space on its work tape.

   Explain why this means that if Reachability is in L, then $\text{L} = \text{NL}$.

   > The configuration graph of a machine running in $O(\log n)$ space is $nc^{\log n}$, so constructing it on the work tape will not work. However, the configuration graph is "static" in that its nodes can be computed one-by-one, without having to modify previously computed values. Thus, we can write the encoded graph directly on the output tape and only use the work tape to keep track of individual transitions. Specifically, we go through all configurations of M, determining the edges in the graph by writing two configurations on the work tape and determining (by looking at the machine description on `input`) whether there is a transition between them. The configurations only need to include the work tape contents, since the input of M is the same as the input of R – and since M uses logarithmic work space, the two configurations on R's work space will also have length $O(\log |x|)$, as required.

> The above algorithm serves as a log-space reduction of a problem S in NL to Reachability, since a nondeterministic machine M accepts a string x in logarithmic space if and only if the configuration graph has a path from the start state to an accepting state, and R produces this configuration graph in logarithmic space. By the property of reductions, if Reachability is in L, the reduction $S \leq_L \text{Reachability}$ implies that $S \in \text{L}$ as well (where we must take care to compose log-space computations without producing intermediate results). Thus, if $\text{Reachability} \in \text{L}$, every problem in NL is in L, proving that $\text{L} = \text{NL}$.

2. Consider the language L in the alphabet $\{a, b\}$ given by $L = \{a^n b^n \mid n \in \mathbb{N}\}$. The language L is *not* in $\text{SPACE}(c)$ for any constant c. Why?

> A Turing machine that uses constant work space is essentially a finite-state automaton: it has a fixed number of cells it can write to, and therefore a fixed limit on the "memory" usage. Indeed, any FA can be converted into a constant-space Turing machine, for example, by turning the set of FA states into the set of TM states, FA transitions into TM transitions, and using no work space (which is certainly in $\text{SPACE}(c)$ for any constant $c \in \mathbb{N}$). If this TM accepted the language $\{a^n b^n \mid n \in \mathbb{N}\}$, so would the original finite automaton; but as you may remember from IA Discrete Mathematics, $\{a^n b^n \mid n \in \mathbb{N}\}$ is *not* a regular language and thus can't be accepted by any finite automaton, leading to a contradiction.

3. Consider the algorithm presented in the lecture which establishes that Reachability is in $\text{SPACE}((\log n)^2)$. What is the time complexity of this algorithm?

   Can you generalise the time bound to the entire complexity class? That is, give a class of functions F such that

$$\text{SPACE}((\log n)^2) \subseteq \bigcup_{f \in F} \text{TIME}(f)$$

> Let n be the length of the path, v be the number of vertices and e be the number of edges. The algorithm goes over, (a) two recursive calls (b) for each vertex (c) each time halving the path-length and (d) a (worst-case) linear search for the edge (a, b).
>
> $$T(n) = \underbrace{2}_{a} \cdot \underbrace{v}_{b} \cdot \underbrace{T(n/2)}_{c} \underbrace{+ e}_{d}$$
>
> The relation can be solved using the usual substitution method:
>
> $$\begin{aligned} T(n) &= 2vT(\tfrac{n}{2}) + e && = 2vT(\tfrac{n}{2}) + e \\ &= 2v(2vT(\tfrac{n}{4}) + e) + e && = 4v^2T(\tfrac{n}{4}) + 2ve + e \\ &= 4v^2(T(\tfrac{n}{8}) + e) + 2ve + e && = 8v^3T(\tfrac{n}{8}) + 4v^2e + 2ve + e \\ &\dots \\ &= (2v)^k T(\tfrac{n}{2^k}) + e \cdot \textstyle\sum_{i=0}^{k-1} (2v)^k && = (2v)^k T(\tfrac{n}{2^k}) + e \cdot O((2v)^k) \end{aligned}$$
>
> where the sum of the geometric series $\sum_{i=0}^{k-1}(2v)^k$ is of the order $O((2v)^k)$. After $k = \log_2 n$

iterations we reach the base case with $T(1) = 1$, giving the sum

$$nv^{\log n} + e \cdot O(nv^{\log n})$$

Since the length of the path must be at most $v - 1$ and the number of edges is at most v^2, this simplifies to

$$n^{\log n+1} + O(n^{\log n+3}) \approx O(n^{\log n+3})$$

A loose bound for the whole complexity class would therefore be

$$\mathrm{SPACE}((\log n)^2) \subseteq \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(n^{k \log n})$$

8. Hierarchy

1. On page 42 of the notes, a number of functions are listed as being constructible. Show that this is the case by giving, for each one, a description of an appropriate Turing machine. Instead of $\lceil \log n \rceil$, you may find it easier to try $n \cdot \lceil \log n \rceil$.

   Prove that if f and g are constructible functions and $f(n) \geq n$, then so are $f \circ g$, $f + g$, $f \times g$ and 2^f.

   n: On input of length n, replace every symbol with 0. This uses linear time and no work space.

   $n \cdot \lceil \log n \rceil$: Read the input one-by-one, erasing the symbol and incrementing a binary counter on the work tape for each cell. Then, write a 0 on the input tape for every symbol on the work tape. The binary counter uses $\lceil \log n \rceil$ work space (so, in fact, $\log n$ is easily shown to be space-constructible), and incrementing it once may require $\lceil \log n \rceil$ bit flips, so the total time for n increments is $n \cdot \lceil \log n \rceil$. By exploiting the fact that each successive bit in the binary representation gets flipped exponentially less often, the time taken to convert the input to binary may be lowered to $O(n) = O(\log n + n)$, which shows that $\lceil \log n \rceil$ is time-constructible as well.

   n^2: Copy the input to the work tape n times, then copy the work tape back to the input, changing all the bits to 0. Space usage is $O(n^2)$, time usage is $O(2n^2) = O(n^2)$.

   2^n: Write 0 on the work tape, then, for each symbol in the input, double the contents of the work tape. Finally, copy the work tape on the input tape. Space and time usage are both exponential.

   $f \circ g$: Assume f and g are constructible and $f(n) > n$. On input of length n, construct g by writing $g(n)$ 0s on the input tape. Construct f using the input $0^{g(n)}$, producing the required string $0^{f(g(n))}$ on the input tape. The work tape can be reused after the construction of g, and since $f(n) > n$, the work space usage of f will dominate with $O(f(g(n)))$. Since the input only needs to be read once, the time usage is $O(n + g(n) + f(g(n)))$, which is also $O(f(g(n)))$.

> $f + g$: Make a copy of the input to a work tape and construct f, not altering the copy (e.g. by writing on a separate work tape). Use the copied input to construct g, concatenating the output to the $0^{f(n)}$ string on the input tape. The space and time used are $O(n+f(n)+g(n))$, but since $f(n) > n$, this is $O(f(n)+g(n))$.
>
> $f \cdot g$: Make a copy of the input to a work tape and construct f, not altering the copy, then use it to construct g, writing its output to a work tape. Finally, copy $0^{g(n)}$ to the input tape $f(n)-1$ more times. Since $f(n) > n$, the space and time usage is $O(f(n) \cdot g(n))$.
>
> 2^f: Construct f, then run the machine that constructs 2^n on $0^{f(n)}$.

2. For any constructible function f, and any language $L \in \mathrm{NTIME}(f(n))$, there is a nondeterministic machine M that accepts L and *all* of whose computations terminate in time $O(f(n))$ for all inputs of length n. Give a detailed argument for this statement, describing how M might be obtained from a machine accepting L in time $f(n)$.

> Let L be a language in $\mathrm{NTIME}(f(n))$ for a constructible function f. By definition, this means that there is a nondeterministing TM M such that for every string $u \in L$ there is at least one accepting computation that uses $O(f(n))$ time, but there is no bound on the runtime of other computations. This question asks us to show that there is a machine M' where the length of *all* computation paths are bounded by f. The construction is similar to Ex. 6.4 in that the bound is explicitly used as a "timer" – the difference being that f is now a constructible function, rather than a polynomial. Even though it's a vastly more general class of functions, the point of the definition of constructible functions is that computing them does not exceed the bound they impose, so using them as a "timer" and/or "yardstick" is appropriate.
>
> The machine M' can be constructed from M by – for example – adding a new tape, and, given an input string u, writing out $f(|u|)$ in unary on the new tape. Then, for every computation step of M, a 1 at the end of the second tape is erased and the tape head is moved one to the left (i.e. the counter is decremented). If the beginning of the timer tape is reached, the string is rejected; otherwise, the result is the same as that of M. By assumption, if u is in L, there must be at least one accepting computation path of M before the timer runs out, so u will be accepted by M' as well. Otherwise, either u is rejected by M before the timer runs out (and so will M'), or the length of the computation exceeds the limit imposed by the timer, in which case the string is again rejected. Since the timer imposes an upper bound on the runtime, no computation path will be longer than $f(|u|)$, and since f is a constructible function, computing the initial timer string on the new tape will take $O(f(|u|))$ time (we can assume that $f(n) \leq n$ for all n since we must at least take n steps to read the input string of length n). Overall, all computations of M' terminate in $O(f(|u|))$ time and the machine accepts L, as required.

Optional exercises

1. POLYLOGSPACE is the complexity class

$$\bigcup_k \text{SPACE}((\log n)^k).$$

   a) Show that, for any k, if $A \in \text{SPACE}((\log \text{n})^k)$ and $B \leq_L A$, then $B \in \text{SPACE}((\log n)^k)$.

   b) Show that there are no POLYLOGSPACE-complete problems with respect to $\leq_L$. (/Hint/: use a) and the Space Hierarchy Theorem).

   c) Which of the following might be true: $\text{P} \subseteq \text{POLYLOGSPACE}$, $\text{P} \supseteq \text{POLYLOGSPACE}$, $\text{P} = \text{POLYLOGSPACE}$?

   d) What is the relationship between the class POLYLOGSPACE and the class Quasi-P (see Exercise Sheet 1, Question 3.1)?

2. In the lecture, a proof of the Time Hierarchy Theorem was sketched. Give a similar argument for the following Space Hierarchy Theorem:

   For every constructible function f, there is a language in $\text{SPACE}(f(n) \times \log(f(n)))$ that is not in $\text{SPACE}(f(n))$.

   Use this to show that if $\text{SPACE}((\log n)^2) \subseteq P$, then $\text{L} \neq \text{P}$.