LTCC Course: Graph Theory 2024/25
Solutions to Exercises 3

Exercise 1.
($\Rightarrow$) Suppose there exists a polynomial $p(n)$ of degree d so that $|f(n)| \leq p(n)$ for all $n \geq 1$. To show that this means $f(n) = O(n^d)$ means that we must show that there exist constants $C > 0$ and $N \geq 1$ so that for all $n \geq N$ we have $|f(n)| \leq Cn^d$. Write $p(n) = a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0$. Then we know $a_d \neq 0$ (since $p(n)$ is of degree d). And in fact we must have $a_d > 0$, otherwise we would have $p(n) \to -\infty$ as $n \to \infty$, which is impossible if $|f(n)| \leq p(n)$ for all $n \geq 1$. So we can estimate for all $n \geq 1$:

$$\begin{aligned} |f(n)| \leq p(n) &= a_d n^d + a_{d-1}n^{d-1} + \cdots + a_1 n + a_0 \\ &\leq a_d n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0| \\ &\leq a_d n^d + |a_{d-1}|n^d + \cdots + |a_1|n^d + |a_0|n^d \\ &= (a_d + |a_{d-1}| + \cdots + |a_1| + |a_0|)n^d. \end{aligned}$$

So by setting $C = a_d + |a_{d-1}| + \cdots + |a_0|$ and $N = 1$, we find $|f(n)| \leq Cn^d$ for all $n \geq N$.
($\Leftarrow$) Suppose we have $f(n) = O(n^d)$, hence there exist constants $C > 0$ and $N \geq 1$ so that for all $n \geq N$ we have $|f(n)| \leq Cn^d$. So for $n \geq N$ the polynomial $p(n) = Cn^d$ would work. But that polynomial might not work for $n < N$. To overcome that problem, set $K = \max\{0, |f(1)|, |f(2)|, \ldots, |f(N-1)|\}$. Then we know that $|f(n)| \leq K$ for all $n < N$. So by setting $p(n) = Cn^d + K$ we are guaranteed $|f(n)| \leq p(n)$ for all $n \geq 1$.

Exercise 2.
These questions are actually harder than they look. I'm not going to give full details of the Turing Machine, but some important ideas that can be used. An important issue for both cases is how to keep track of what part of the string has already been copied or doubled, and what part still needs to be done. Although it seems tempting to use a "counter" to indicate what part needs to be copied/doubled next, such a counter is actually quite hard to implement. So it's probably much more convenient to use a "marker" to indicate where the machine was. Such a marker can just be a square on the tape with the blank symbol # on it.

Of course, that means the machine erases what was on that square before. So somehow the machine needs to store that information. Since there are only two possible original symbols (0 or 1), this can be done using appropriate states. So there should be a set of states just for when the machine is dealing with a 0, and a set of states for when the machine is dealing with 1.

So in the sketches of the two Turing Machines below, we divide the state space Q into three parts: $Q = Q' \cup Q^0 \cup Q^1$. Here Q' contains some general states (including q_0 and so), while Q^0 and Q^1 are sets of states the machine can be in while it is copying a 0 or a 1, respectively.

(A) Apart from the idea to use the blank symbol as a marker, another observation that makes life a lot easier for this problem is to start working from the back of the string.

To understand the steps of the Turing Machine as given below, you should have an idea how the tape is used. So here is how the tape would look like in the middle of the computation of making a copy of the string 00010001011:

$$\cdots|\#|\#|1|0|1|1|\#|\underline{0}|0|0|1|0|\blacksquare\,0\,|\#|1|0|1|1|\#|\#|\cdots .$$

The underlined square is the starting square, while the black blob is the tape head. So the machine has just erased the symbol on the square to the right of the tape head. That square was erased, but since it contained originally the symbol 0, the machine entered in one of the states from Q^0, and the tape head started to move to the left.

The Turing Machine performs steps as follows, all the time using states from Q^0:
– The tape head goes left to the beginning of the original string (recognised by the first blank it encounters).
– Then it moves further left to the beginning of the already copied part (again, recognised by a blank square).
– Once it encounters that blank square, it writes a 0, and starts moving right again.
– The tape head goes right till the beginning of the original string (recognised by a blank).
– Then further right till the marker of the bit it was working on (again a blank).
– Then it writes a 0 to replace the blank (so that square is back to the original state). It also moves one square to the left, and the state becomes one of the non-specific states from Q'.
– If it now reads a blank symbol, the copying is done, and the machine halts. If it doesn't read a blank symbol, it replaces the current symbol by the blank symbol; and its state becomes a state from Q^0 or Q^1, depending if the square contained a 0 or a 1.
– And now the process starts as from the first step above.

With the description above, it should be possible to design a full Turing Machine. You probably need a few initial steps to get going and to make sure that the machine halts immediately if the empty string is given as input.

(B) Several of the ideas from A can be used here as well. Here an extra complication is to make room for the bits that have to be added in between. The best way to do this is probably by doing it one by one. I.e. every time when making a copy of the next bit, first move the remainder of the string one to the right (starting at the back again), and then make the copy in the freed space.

Let's demonstrate the steps and the main ideas by looking at what the situation could be in the middle of a computation again. Using the same string 00010001011 as example again, a typical situation on the tape in the middle of the computation will be something like:

$$\cdots|\#|\#|\underline{0}|0|0|0|0|1|1|\#|\blacksquare\,0\,|0|1|0|1|1|\#|\#|\cdots .$$

So the machine has already doubled the first four bits, and has just erased the fifth bit, and moved one step to the right. Since that erased symbol was a 0, the machine will be in some state from Q^0.

And next the machine will perform the following steps:
– It will move right to the end of the string (recognised by a blank square).
– It then will do one step back to the left, read the symbol, erase it, move one step to the right, write that one symbol on that square, and one step to the left again. (These steps are just moving one symbol one square to the right.)
– It continues doing the above until it is back to the beginning of the part of the original string that hasn't been doubled yet.

– Once it is back at the blank "marker", there will actually be two blank squares in a row (since the complete right side of the string has been moved one square further to the right). So it can write two 0s in a row. (The machine "knows" these must be 0s since it was using states from Q^0 in all the steps above.)
– The machine moves one square further to the right, and the state becomes one of the non-specific states from Q'.
– If it now reads a blank symbol, the doubling is done and the machine halts. If it doesn't read a blank symbol, it replaces the current symbol by a blank, and goes into a state from Q^0 or from Q^1, according to the symbol that was on the tape.
– And now the process starts as from the first step above.

Again, the final Turing Machine probably needs some extra initial steps to get it all going and to make sure it halts if there is nothing on the tape. But apart from that, you should be able to design a Turing Machine that does all of the above.

Exercise 3.
(a) By definition, L_1 is in NP, if there exists a nondeterministic Turing Machine $\mathcal{N}^{(1)}$ and a polynomial $p^{(1)}(n)$ such that for every $x \in L_1$, there exists a sequence of at most $p^{(1)}(|x|)$ allowed steps so that $\mathcal{N}^{(1)}$ halts in the accepting state of $\mathcal{N}^{(1)}$. And a similar conclusion, using a nondeterministic Turing Machine $\mathcal{N}^{(2)}$ and a polynomial $p^{(2)}(n)$, can be drawn from the claim that L_2 is in NP.

From this we can easily see that also $L_1 \cup L_2$ is in NP. Make a new nondeterministic Turing Machine $\mathcal{N}$ that is formed by using disjoint states $q_i^{(1)}$ for $\mathcal{N}^{(1)}$ and states $q_i^{(2)}$ for $\mathcal{N}^{(2)}$. Now let q_0 and q_Y be two new states. Then form $\mathcal{N}$ as follows. Starting at state q_0, there is one possible transition to the initial state $q_0^{(1)}$ of $\mathcal{N}^{(1)}$ and one possible transition to the initial state $q_0^{(2)}$ of $\mathcal{N}^{(2)}$. Also, from the accepting state $q_Y^{(1)}$ of $\mathcal{N}^{(1)}$ there is one transition to the accepting state q_Y of $\mathcal{N}$, and the same holds for the accepting state of $\mathcal{N}^{(2)}$.

It's now straightforward to check that if $x \in L_1 \cup L_2$, then the nondeterministic Turing Machine $\mathcal{N}$ has a sequence of at most $\max\{p^{(1)}(|x|), p^{(2)}(|x|)\} + 2$ allowed steps so that $\mathcal{N}$ halts in its accepting state. And if $x \notin L_1 \cup L_2$, then $x \notin L_1$ and $x \notin L_2$, so with input x, $\mathcal{N}$ can never reach the accepting state of the $\mathcal{N}^{(1)}$ and the $\mathcal{N}^{(2)}$ part of $\mathcal{N}$, and hence such an x will never be accepted.

(b) We use the set-up from (a) again. And in fact we can use almost the same ideas, except now we construct a nondeterministic Turing Machine $\mathcal{N}$ as follows: From the initial state q_0 there is one transition to the initial state $q_0^{(1)}$ of $\mathcal{N}^{(1)}$; from the accepting state $q_Y^{(1)}$ of $\mathcal{N}^{(1)}$ there is one transition to the initial state $q_0^{(2)}$ of $\mathcal{N}^{(2)}$; and from the accepting state $q_Y^{(2)}$ of $\mathcal{N}^{(2)}$ there is one transition to the general accepting state q_Y. (For this, we need to assume in addition, that after the computations of $\mathcal{N}^{(1)}$ the tape again just contains the original input; but it is easy to modify $\mathcal{N}^{(1)}$ accordingly by making it first copy the input and erase all intermediate calculations at the end.)

Again when we give an input $x \in L_1 \cap L_2$ to the resulting nondeterministic Turing Machine $\mathcal{N}$, there is a sequence of at most $p^{(1)}(|x|) + p^{(2)}(|x|) + 3$ allowed transitions after which $\mathcal{N}$ has reached its accepting state. And for $x \notin L_1 \cap L_2$, at least one of the two parts will prevent that from happening.

(c) This time we can't conclude that $L_1 \setminus L_2$ is in NP. The reason is that the fact that $x \in L_1 \setminus L_2$ means that $x \in L_1$ and $x \notin L_2$. But we have no good way to deal with the situation $x \notin L_2$.

In general we don't know if $L_1 \setminus L_2$ is in NP or not. If P = NP, then the statement is true (because then NP = P = coP = coNP), but it's generally believed that coNP is not equal to NP. If that's the case, then take L_2 to be a language in NP but not in coNP (which exists by symmetry) and L_1 to be the trivial language containing all words (which is in NP), and we see that $L_1 \setminus L_2$ is a language in coNP but not in NP.

Exercise 4.
To show that $\leq_P$ is a quasi-order, we have to show that (a) $\mathcal{L} \leq_P \mathcal{L}$ for every language $\mathcal{L}$, and (b) if $\mathcal{L}_1 \leq_P \mathcal{L}_2$ and $\mathcal{L}_2 \leq_P \mathcal{L}_3$, then $\mathcal{L}_1 \leq_P \mathcal{L}_3$.

For (a), the function f in the formal definition of $\leq_P$ is the identity: informally, if we have a method to determine membership in $\mathcal{L}$, then we can use this method trivially to determine membership in $\mathcal{L}$. For (b), suppose we are given functions $g, h : \{0,1\}^* \to \{0,1\}^*$ so that g certifies $\mathcal{L}_1 \leq_P \mathcal{L}_2$ and h certifies $\mathcal{L}_2 \leq_P \mathcal{L}_3$. Then $x \in \mathcal{L}_1 \Leftrightarrow g(x) \in \mathcal{L}_2 \Leftrightarrow hg(x) \in \mathcal{L}_3$. Moreover, since both g and h can be computed by Turing machines in polynomial time, so can the composition hg. Hence hg certifies that $\mathcal{L}_1 \leq_P \mathcal{L}_3$.

Now let $\mathcal{L}$ be an element of P. To show that $\mathcal{L}$ is a minimal, we need to show that $\mathcal{L} \leq_P \mathcal{L}'$ for every non-trivial language $\mathcal{L}'$. Because $\mathcal{L}'$ is non-trivial, there is some word w_1 that is in $\mathcal{L}'$, and some word w_0 that is not. To show that $\mathcal{L} \leq_P \mathcal{L}'$, consider the function f that maps all words in $\mathcal{L}$ to w_1, and all words not in $\mathcal{L}$ to w_0. This function can be computed in polynomial time by a Turing machine, exactly because $\mathcal{L}$ is in P (more details are given below, if you are interested).

Some more details for showing $\mathcal{L} \leq_P \mathcal{L}'$: If $\mathcal{L}$ is in P, then there is a Turing Machine $\mathcal{M}$ and an integer d so that for all $x \in \{0,1\}^*$ we have: if $x \in \mathcal{L}$, then $\mathcal{M}$ accepts x in $O(|x|^d)$ steps; while if $x \notin \mathcal{L}$, then $\mathcal{M}$ rejects x after $O(|x|^d)$ steps.

Now define the function $f : \{0,1\}^* \to \{0,1\}^*$ as follows: $f(x) = \begin{cases} y_1, & \text{if } x \in \mathcal{L}; \\ y_2, & \text{if } x \notin \mathcal{L}'. \end{cases}$

Since $y_1 \in \mathcal{L}'$ and $y_2 \notin \mathcal{L}'$, it's obvious that for all $x \in \{0,1\}^*$ we have $x \in \mathcal{L}$ if and only if $f(x) \in \mathcal{L}'$. Moreover, using the Turing Machine $\mathcal{M}$ from above, for all $x \in \mathcal{L}'$, $f(x)$ can be computed in polynomial time. (Just give as output y_1 if $\mathcal{M}$ accepts x, and as output y_2 if $\mathcal{M}$ rejects x.) This shows $\mathcal{L} \leq_P \mathcal{L}'$.

Some extra notes: Although the above answer is (hopefully) not too hard to follow, there are some deeper issues it touches. In particular, the algorithm described assumes we have knowledge of suitable y_1 and y_2. By the formulation of the question, we can be sure suitable y_1, y_2 exist. And hence the algorithm exists.

But if we would like to implement the algorithm above, we would need to know explicitly what y_1 and y_2 are. And that could be a problem (or a lot of work).

So keep this in mind: The definitions of P and the other complexity classes talk about the existence of certain Turing Machines. It doesn't require that you are actually able to construct them.

Exercise 5.
Property 7 states: *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 fact that $\mathcal{L}_1$ is NP-complete means that $\mathcal{L}_1$ is in NP, and for all $\mathcal{L} \in$ NP we have that $\mathcal{L} \leq_P \mathcal{L}_1$. And to prove that $\mathcal{L}_2$ is NP-complete we need to prove that $\mathcal{L}_2$ is in NP, and for all $\mathcal{L} \in$ NP we have that $\mathcal{L} \leq_P \mathcal{L}_2$.

We are already given that $\mathcal{L}_2$ is in NP, so we are left to prove that $\mathcal{L} \leq_P \mathcal{L}_2$ for all $\mathcal{L} \in$ NP. For this we obviously need to use that $\mathcal{L} \leq_P \mathcal{L}_1$ for all $\mathcal{L} \in$ NP and that $\mathcal{L}_1 \leq_P \mathcal{L}_2$. In fact it follows directly that it is enough to prove the following property:

For any three languages $\mathcal{L}, \mathcal{L}', \mathcal{L}''$, if $\mathcal{L} \leq_P \mathcal{L}'$ and $\mathcal{L}' \leq_P \mathcal{L}''$, then $\mathcal{L} \leq_P \mathcal{L}''$.

But this is the transitivity property we already shows in the solution of Exercise 4.

Exercise 6.
Property 8 states: *Every non-trivial language in P is P-complete.*

Let $\mathcal{L}' \subseteq \{0,1\}^*$ be a language in P, $\mathcal{L}' \neq \varnothing$ and $\mathcal{L}' \neq \{0,1\}^*$. In particular this means that there are $y_1, y_2 \in \{0,1\}^*$ so that $y_1 \in \mathcal{L}'$ but $y_2 \notin \mathcal{L}'$. We need to show that for all $\mathcal{L} \in$ P we have $\mathcal{L} \leq_P \mathcal{L}'$, which means that $\mathcal{L}'$ is P-complete. But we already showed this in the solution of Exercise 4.

Exercise 7.
The easiest way to show 3-SAT is NP-complete is to reduce SATISFIABILITY to it and apply Proposition 8. It's obvious 3-SAT is in NP as it's a special case of SATISFIABILITY. So all we need to do is figure out how to go from a SATISFIABILITY formula (which we will assume is in CNF form, since that's the specific form we proved is NP-complete in Theorem 9) to a 3-SAT formula.

We can replace a clause (x, y) with $(x, y, t)(x, y, \bar{t})$ where t is a new variable. The point is, whether t is True or False, to satisfy both clauses we need at least one of x and y to be True. We can do something similar (with two new variables and four 3-SAT clauses) to deal with a clause (x).

So it remains to deal with long clauses that have four or more literals in them. We can replace (x, y, z, w) with $(x, y, t)(z, w, \bar{t})$, where t is again a new variable (which we use only in these two clauses). More generally, we can repeat this trick to split any k-variable clause into two clauses of sizes $\lfloor \frac{1}{2}k \rfloor + 1$ and $\lceil \frac{1}{2}k \rceil + 1$ respectively, which we can split again if necessary and so on.

The reduction in the last paragraph above doesn't work for 3-SAT. The point where it goes wrong is that when we try to split a 3-literal clause we get a 2-literal and a 3-literal clause; so we are no better off than when we started.

There are several ways to solve 2-SAT instances in polynomial time. One way is simply to pick a variable x and use the following procedure. Set x to True. For each clause containing $\overline{x}$, set the other variable to True or False in order to satisfy the clause, and keep doing this until either a variable becomes conflicted (i.e. that variable needs to be set True to satisfy one clause but False for another), or no further variables are forced. If the setting procedure terminates without finding a conflict, then it has set a collection X of variables. Now any clause which contains a False literal belonging to one of these variables also (by the setting procedure) contains a True literal from one of these variables; in other words, any clause containing literals from the variables X is now satisfied, and we can remove all of them.

If what remains is unsatisfiable, the original formula obviously was too; if what remains is satisfiable, then the variables X can take any value in a satisfying assignment, in particular the one given by the setting procedure, so the original formula was also satisfiable.

So the full algorithm is: iteratively, pick an un-set x. Try setting it True and following the setting procedure; if there is no conflict discovered, move on to the next un-set variable. If a conflict is discovered, instead set x False and run the same setting procedure. If there is again a conflict, the formula is not satisfiable. If not, move on to the next un-set variable. At the end, either all variables are set and we have a satisfying assignment, or the formula is not satisfiable.

Exercise 8.
It is obvious that HALF-STABLE-SET is in NP: the certificate would be a set of vertices; it is easy to check if this is of size $\frac{1}{2}|V|$ and if it is an independent set.

We can reduce STABLE-SET to HALF-STABLE-SET as follows. Given an instance (G, K) of STABLE-SET, we can add vertex sets I and C to G, where we let I be an independent set with no edges to $V(G)$, and let the vertices of C be adjacent to all other vertices. The resulting graph has an independent set of size $K+|I|$ if and only if G has an independent set of size K. To get an instance of HALF-STABLE-SET we need to choose $|I|$ and $|C|$ so that $K + |I| = \frac{1}{2}(|V(G)| + |I| + |C|)$; i.e. so that $|I| - |C| = |V(G)| - 2K$.

Exercise 9.
It is easy to see that there is a cycle in G through u if and only if there is an edge uv and a path from u to v in the graph G' formed by removing the edge uv from G. Using Savitch's Theorem 13 from Section 3.10 of the notes, it is fairly easy to construct a way to do this checking for one edge uv in the same amount of space it requires to check ST-CONNECTIVITY (plus a little bit extra to store the edge uv we are currently dealing with). We then simply have to try all the different neighbours v of u, one after the other (erasing the working take in between).

More explicitly, build a 2-tape Turing Machine that does the following. (Here we again assume that the vertices are indicated by the numbers 1 to N.)
1. Write the vertices u and v on the tape and set $a = 1$.
2. Write a on the tape.
3. Read the input tape to check if ua is an edge.
- If ua is an edge, continue with step 4 below.
- If ua is not an edge and $a = N$, then conclude that there is no cycle through u and stop.
- If ua is not an edge and $a < N$, then increase a by one and go beck to step 2.
4. Use Savitch's Algorithm to check if there is a path from u to a in G, where we will ignore the edge ua in the computation.
- If there is such a path, then conclude that there is a cycle through u and stop.
- If such a path does not exist and $a = N$, then that there is no cycle through uv; then we should repeat the procedure with the next neighbour v; if v is the last neighbour, then there is no cycle through u and we stop.
- If such a path does no exist and $a < N$, then increase a by one and go beck to step 2.

Since step 4 can be done using at most $O((\log(N))^2)$ memory space, we can easily see that the whole procedure can be done using at most $O((\log(N))^2)$ memory space.