Complexity Theory

Supervision 1 – Solutions

1. Algorithms and problems

1. a) Explain the formal connections between the notions of *characteristic function*, *predicate*, *decision problem*, *subset* and *language*.

> All five notions refer to the same set-theoretic concept in different settings. The most fundamental notion is that of a *subset* of a set: $S \subseteq A$ if $\forall a \in S.\ a \in A$. In formal language theory, when A is a set of strings Σ^* for an alphabet Σ, a subset L of Σ^* is called a *language*. A *decision problem* is a question on some input that has a yes-no answer; the typical decision problem associated with a language $L \subseteq \Sigma^*$ is deciding whether a string $s \in \Sigma^*$ is in L or not. Set-theoretically, a decision problem is a *predicate*, i.e. a Boolean-valued function. The decision problem associated with a language L is therefore a predicate of type $\Sigma^* \to \mathbb{B}$, and is precisely the *characteristic function* χ_L for the subset $L \subseteq \Sigma^*$. In summary, a language is a subset of the set of all possible strings of an alphabet, every subset comes with a characteristic function that captures the decision problem of set membership, and a decision problem corresponds to a particular predicate on the set of strings.

   b) What is the difference between a Turing machine *accepting* vs. *deciding* a language L? How does this distinction relate to the difference between *recursively enumerable* and *decidable* languages? (Note: instead of *accepting*, *decidable* and *recursively enumerable*, you will also often see the terms *recognising*, *recursive* and *semidecidable*, respectively).

> A TM *deciding* a language L must halt on every input $s \in \Sigma^*$ and return one of two answers (`acc` or `rej`) in some finite number of computation steps. That is, the TM will say "yes" if $s \in L$, and will say "no" if $s \notin L$. In contrast, a TM *accepting* a language L does not have to halt on every input $s \in \Sigma^*$, but it *does* halt for strings for which there is a computation ending in the accepting state. That is, the TM will say "yes" if $s \in L$, but will not necessarily say "no" if $s \notin L$.
>
> Languages accepted by a TM are called semidecidable or recursively enumerable (a Turing machine can enumerate all the strings in the language), while languages decided by a TM are called decidable or recursive. Decidable languages are semidecidable, but not vice versa: for instance, the Halting Problem is undecidable, but it *is* semidecidable (the universal Turing machine halts if its input halts, and diverges otherwise, which is exactly the definition of acceptance).

   c) What set-theoretic object (what kind of function or relation) is implemented by a Turing machine accepting vs. deciding a language?

> A TM deciding a language $L \subseteq \Sigma^*$ implements the (total) characteristic function $\chi_L : \Sigma^* \to \mathbb{B}$, since it must be defined for any input. On the other hand, accepting a language corresponds to a *partial* predicate $\Sigma^* \rightharpoonup \mathbb{B}$, since the output is not guaranteed to be defined. In a sense, we are working with "three-valued logic"; indeed, as partial functions $A \rightharpoonup B$ can be represented as total functions $A \to B \uplus \mathbf{1}$ for the one-element set $\mathbf{1}$, and $\mathbb{B} \cong \mathbf{1} \uplus \mathbf{1} = \mathbf{2}$, acceptance corresponds to a total, three-valued predicate $\Sigma^* \to \mathbf{3}$.

2. Say we are given a set $V = \{v_1, \ldots, v_n\}$ of vertices and a cost matrix $c : V \times V \to \mathbb{N}$. For an index $i \in [1..n]$ and a subset $S \subseteq V$, let $T(S, i)$ denote the cost of the shortest path that starts at v_1, and visits all vertices in S, with the last stop being $v_i \in S$. Describe a dynamic programming algorithm that computes $T(S, i)$ for all sets $S \subseteq V$ and all $i \leq |V|$. Show that your algorithm can be used to solve the Travelling Salesman Problem in time $O(n^2 2^n)$.

   The dynamic programming implementation of this algorithm relies on the following recursive optimality condition: every subpath of an optimal subpath is itself optimal. In other words, the travelling salesman takes all the possible shortcuts they can, so part of their journey will be the shortest of all possible journeys between the same endpoints. Thus, we can set up our dynamic programming solution by identifying a subproblem that is indexed by something we can do induction on, and which has our problem as a special case (cf. Kleene's definition of a regular expression for a given DFA). As the question suggests, such a subproblem is computing $T(S, i)$ for a subset S and index i, which is the cost of the shortest path $v_1 \to^* v_i$ that goes through all vertices in S. First, the sanity check: does this cover the original travelling salesman problem? Indeed it does: the overall cost of the minimal cycle will simply be $T(V, 1)$, i.e. the cost of the shortest path that starts at v_1, visits all the vertices, and ends back at v_1.

   The next problem is actually computing $T(S, i)$, which we can do by recursively calling it on smaller subproblems, i.e. smaller sets of intermediate nodes. The base cases will be when S contains the endpoint only (we cannot end at a node without traversing it, so $T(S, i)$ will always be infinite cost when $v_i \notin S$): then, the cost $T(\{v_i\}, i)$ of going from v_1 to v_i is simply the distance $c(v_1, v_i)$ of the nodes. To compute the general case $T(S, i)$, consider an intermediate point $v_x \in S \setminus \{v_i\}$. Due to the optimality condition, the shortest path to v_i will consist of the shortest path to a particular v_x through all the nodes in S other than v_i, plus the cost of continuing from v_x to v_i in one step. The v_x that we need is the one that will minimise this sum:

$$T(S, i) = \min_{v_x \in S \setminus \{v_i\}} (T(S \setminus \{v_i\}, x) + c(v_x, v_i))$$

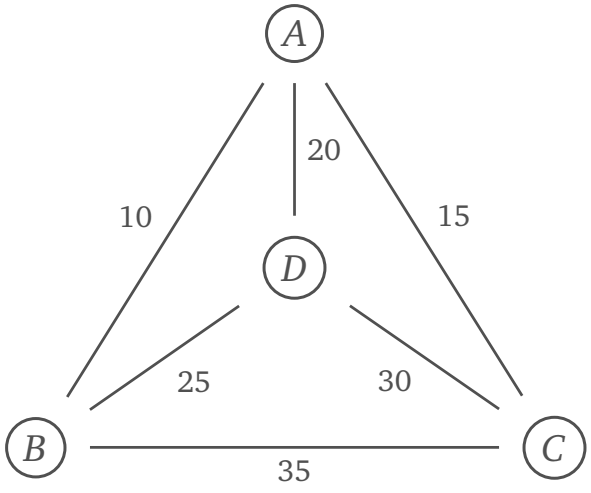
   With these definitions, we are ready to fill out the memoisation table for the dynamic programming solution. The "coordinates" will be given by the possible values of S and i: there will be n columns (one for each vertex), but 2^n rows (one for each possible subset S). Note that the table is likely to be sparse, since the entry for S and i will be finite only

if $v_i \in S$ – however, trying to optimise for this will not result in an asymptotically better algorithm. We start filling out the table with the base cases $S = \{ v_i \}$ for each index i. To compute the cost of an arbitrary cell $T(S, i)$, we look through all the n entries in the row at $S \setminus \{ v_i \}$ and find the minimum one when added to the cost of getting from there to v_i. Once the table is complete, the value we are looking for is in the bottom left corner, for $S = V$ and $i = 1$.

This high-level implementation gives us the time and space complexity of the algorithm directly. Space use is the size of the memoisation table, $O(n2^n)$. The time complexity is the number of entries which we need to compute times the cost of computing each entry. To find a cell, we perform a linear minimisation algorithm for n previous entries, and we do this for $n2^n$ cells – thus, the overall time complexity is $O(n^2 2^n)$, as required.

As a concrete example, consider the graph below with its memoisation table (where the empty entries correspond to ∞). For example, to compute the second (**B**) entry in the A, B, C row, we look at the A, C row and take the minimum of $30 + c(A, B) = 40$ and $15 + c(B, C) = 45$. The bottom-left entry corresponds to visiting all the nodes and ending at A. By also keeping track of the partial paths, we could output $A \to C \to D \to B \to A$ alongside the total weight 80.

	A	**B**	**C**	**D**
$\emptyset$				
A	0			
B		10		
C			15	
D				20
A, B	20	10		
A, C	30		15	
A, D	40			20
B, C		50	45	
B, D		45		35
C, D			50	45
A, B, C	60	40	35	
A, B, D	55	45		35
A, C, D	65		50	45
B, C, D		70	75	75
A, B, C, D	**80**	70	65	65



3. The lectures define Turing machines to have a transition function of type $\delta : (Q \times \Sigma) \to (Q \cup \{ \texttt{acc}, \texttt{rej} \}) \times \Sigma \times D$, where $D = \{ L, R, S \}$ is the set of directions that the tape head can move in (left, right, stationary). In other literature you might find definitions that have $D = \{ L, R \}$, allowing only two directions and requiring that the tape head move at each transition. Can such a Turing machine simulate the one described in lectures? Is the complexity class P affected by this distinction?

> The translation is indeed possible: we can replace a stationary direction with a "back-and-forth" move to the right and then back to the left. However, to make this unambiguous, we have to explicitly mark when we are halfway through this zig-zag, and restore the appropriate state afterwards. We cannot do this without doubling the number of states: for every $q \in Q$, we need to also add a copy q' that we transition to from q with the right move, and then go from q' to the state we would have ended up in with the stationary move on the left transition. In summary, instead of including the δ-mapping $(p, a) \mapsto (q, b, S)$, we add mappings $(p, a) \mapsto (q', b, R)$ for all Q and $a \in \Sigma$, and $(q', c) \mapsto (q, c, L)$ for all $q \in Q$ and $c \in \Sigma$. The second part of the definition also ensures that before moving back to the left, we don't alter the symbol under the tape head.
>
> This transformation doubles the number of states used, and may make the computation twice as long (in the worst case). Of course, this does not make polynomial-time algorithms run exponentially – the class P does not change. There are many other variations on the formal definition of a Turing machine which can all be shown to be equivalent to the definition in the notes despite, perhaps, seeming more general and powerful. For instance, we can work with TMs with a two-way infinite tape, with multi-track tapes, with multiple tapes – all of them can be implemented by the simplest TM with an at most polynomial slowdown.

2. Polynomial-time problems

1. Consider the language $\mathrm{Unary\text{-}Prime}$ in the one-letter alphabet $\{a\}$ defined by $\mathrm{Unary\text{-}Prime} = \{a^n \mid n \text{ is prime}\}$. Show that this language is in P.

   > This question may seem surprising considering the discussion of PRIME in the notes. However, there is a very important difference: PRIME takes the *binary encoding* of a number n as its input, and the brute-force algorithm of checking for divisors in $\sqrt{n}$ steps is indeed exponential in the *length* of the binary encoding (but not the number n itself). If the size of the input (which is what we care about when analysing the complexity of algorithms) is x, the largest number it may encode is $2^x - 1$; the number of steps taken by the naive algorithm is approximately $\sqrt{2^x} \approx 1.414^x$, which is indeed exponential in the size of the input.
   >
   > The language $\mathrm{Unary\text{-}Prime}$ differs in that it uses *unary* encoding: the number corresponding to an input of size x is exactly x. Now, the naive algorithm runs in $x^{\frac{1}{2}}$ steps, which is polynomial in the size x – hence, $\mathrm{Unary\text{-}Prime}$ is in P. This shows that while many of the internal details of the Turing machine do not matter (number of tapes, possible directions, etc.), the *input encoding* used makes a significant difference.

2. Suppose $S \subseteq \mathbb{N}$ is a subset of natural numbers and consider the language $\mathrm{Unary}\text{-}S$ in the one-letter alphabet $\{a\}$ defined by $\mathrm{Unary}\text{-}S = \{a^n \mid n \in S\}$, and the language $\mathrm{Binary}\text{-}S$ in the two-letter alphabet $\{0, 1\}$ consisting of those strings starting with a 1 which are the binary representation of a number in S. Show that if $\mathrm{Unary}\text{-}S$ is in P, then $\mathrm{Binary}\text{-}S$ is in $\mathrm{TIME}(2^{cn})$

for some constant c.

> As explained in the previous question, using unary encoding gives us some "legroom" when calculating the complexity of the algorithm. If we have a polynomial algorithm deciding S that runs in $O(n^c)$ time for some $n, c \in \mathbb{N}$ a Turing machine using unary encoding will also run in polynomial time in the size n of the input. However, using binary encoding, the size of the input x will only be $\log n$. The same algorithm will now run in $O((2^x)^c) = O(2^{cx})$ time in the size x, which is in $\mathrm{TIME}(2^{cn})$.

3. We say that a propositional formula φ is in 2CNF if it is a conjunction of clauses, each of which contains exactly two literals. The point of this problem is to show that the satisfiability problem for formulas in 2CNF can be solved by a polynomial time algorithm.

   First note that any clause with two literals can be written as an implication in exactly two ways. For instance $(P \vee \neg Q)$ is equivalent to $(Q \implies P)$ and $(\neg P \implies \neg Q)$, and $(P \vee Q)$ is equivalent to $(\neg P \implies Q)$ and $(\neg Q \implies P)$. For any formula φ, define the directed graph G_φ to be the graph whose set of vertices is the set of all literals that occur in φ, and in which there is an edge from literal P to literal Q if, and only if, the implication $P \implies Q$ is equivalent to one of the clauses in φ.

   a) If φ has n variables and m clauses, give an upper bound on the number of vertices and edges in G_φ.

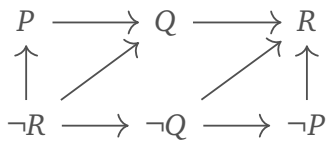
   > Since each clause can be converted to two different implications, every variable will correspond to both a nonnegated and negated graph vertex – thus, for n variables, the graph will contain $2n$ vertices. Similarly, the number of edges will be $2m$, twice the number of clauses. Since some transformations may lead to repetition (e.g. we might have a clause $\{P, Q\}$ and a clause $\{\neg P, \neg Q\}$ which give the same pair of implications), these are both upper bounds.

   b) Show that φ is *unsatisfiable* if, and only if, there is a literal P such that there is a path in G_φ from P to $\neg P$ *and* a path from $\neg P$ to P.

   > ($\Leftarrow$): Suppose there are two paths $P \to^* \neg P$ and $\neg P \to^* P$ in G_φ. Since the edges correspond to implications and implication is transitive, two such paths would establish the bi-implication $P \iff \neg P$, which is of course a contradiction for any P. By the construction of G_φ, this shows that the clauses imply a contradiction and are therefore unsatisfiable. More precisely, if there was a satisfying interpretation that assigned P to be true (or false, in which case the reasoning is similar), the path from P to $\neg P$ would have to include at least one edge that switches from true to false, corresponding to the implication $\top \implies \bot$ generated from a clause $\bot \vee \bot = \bot$. This contradicts our assumption that our original interpretation satisfied the conjunction of clauses, so such an interpretation cannot exist.
   >
   > ($\Rightarrow$): To show the converse, we prove the contrapositive statement: if there does

not exist a two-way path between opposite literals, we can always find a satisfying interpretation. While it seems like proving the original statement is easier (we often do proof by contrapositive to *avoid* having to prove existentials), this direction actually gives us a constructive proof of the existence of the satisfying interpretation. There are of course other approaches, such as Kosaraju's algorithm for finding strongly connected components in the graph, and checking if contradicting pairs of literals appear in a single SCC.

Suppose the graph G_φ does not have any two-way paths between opposite literals; as an example, consider the graph below, corresponding to the clause set $\{Q, R\}, \{Q, \neg P\}, \{\neg Q, R\}, \{R, P\}$.



Choose a vertex P which doesn't yet have a truth value assigned to it, and there isn't a path from P to $\neg P$. This is always possible, since if a node P implies its negation, by our original assumption $\neg P$ does not imply P so we can just start with the vertex corresponding to $\neg P$. Next, set all literals Q implied by P to true (that is, all vertices Q such that $P \to^* Q$), and all the negations of these Q in the rest of the graph to false. We repeat this process until all vertices have a truth value assigned to them, and this will give us a satisfying interpretation of the original set of clauses.

There are two things to check: that the assignment will never encounter conflicts, and that at the end of the algorithm we actually get a satisfying interpretation. The latter is a simple consequence of our construction of G_φ and the main algorithm step: a true node can only ever imply a true node, so there will be no "forbidden" edges $\top \implies \bot$ that would falsify the set of clauses. We also need to make sure that the transitive assignment of truth values will never result in conflicts, i.e. there will never be cases where P implies both Q and $\neg Q$. Suppose there is a path $P \to^* Q$ and, for contradiction, assume there is another path $P \to^* \neg Q$. There is a contrapositive path $Q \to^* \neg P$ corresponding to the second assumption, since both are generated from the same clause $\neg P \vee \neg Q$. But then we can go from P to $\neg P$ via $P \to^* Q \to^* \neg P$, which contradicts our original constraints in choosing the vertex P. Thus, no algorithm step will result in conflicts and we will always be able to find assignments for every vertex, which gives us a satisfying interpretation of the original set of clauses.

c) Give an algorithm for verifying that a graph G_φ satisfies the property stated in (b) above. What is the complexity of your algorithm?

The method described above shows that we can find a satisfying implementation when the graph has no contradictions. It can be adapted into a decision procedure which either determines that a set of clauses is unsatisfiable, or gives a satisfying

> interpretation. It is essentially the above procedure implemented as a backtracking algorithm. We start with an arbitrary vertex without a truth assignment, and set every vertex it implies to true, and the negations of those vertices to false. If we ever reach a conflict (P implying $\neg P$ or both Q and $\neg Q$), we backtrack and start from a different node. If we manage to fill in the whole graph, we output the satisfying interpretation. If upon backtracking there are no other nodes we can try, but not all nodes have a truth assignment, we must have an implication graph with a two-way path between opposite literals (as a consequence of the proof above), so the original set of clauses is unsatisfiable.
>
> The time complexity of this algorithm is $O(n+m)$, though the precise runtime depends on the implementation of backtracking. Other approaches (such as finding SCCs) also have the same time complexity.

d) From (c) deduce that 2CNF-SAT is in P.

> The input to the problem would be some representation of the graph (such an adjacency list or matrix), but in either case, an $O(n+m)$ algorithm would be polynomial in the size of the input (in fact, linear), so 2CNF-SAT is in P.

e) Why does this idea not work if we have three literals per clause?

> The edges of the implication graph are constructed from clauses containing two literals – a 3CNF clause cannot be interpreted as a single edge.

4. A clause (i.e. a disjunction of literals) is called a *Horn clause* if it contains at most one positive literal. Such a clause can be written as an implication: $X \vee \neg Y \vee \neg W \vee \neg Z$ is equivalent to $(Y \wedge W \wedge Z \implies X)$. HORNSAT is the problem of deciding whether a given Boolean expression that is a conjunction of Horn clauses is satisfiable.

   Show that there is a polynomial time algorithm for solving HORNSAT.

> The algorithm we are looking for is nothing but DPLL – in fact, we don't even need the full power of DPLL, since the Horn clause constraint makes its more expensive steps unnecessary. We start with unit propagation: for any clause $\{ L \}$, we perform the appropriate variable assignment that makes L true (i.e. if $L = \neg P$, we set P to $\bot$, otherwise we set it to $\top$), delete any clause that contains L and remove $\neg L$ from the remaining clauses. We continue this until there are no unit clauses left. At this point, DPLL would require us to do a case split; however, with Horn clauses, we know that every clause must contain at least one negated literal (since Horn clauses can only have at most one positive literal, and we have no unit clauses). To satisfy all the remaining clauses, we simply pick a negated literal from each, and set the associated variables to $\bot$. As a result, every clause will contain a literal $\neg\bot$, making all of them true. This process gives us a satisfying interpretation, if it exists; otherwise, at some point, we would end up with the empty clause $\{\ \}$, signifying that the clauses are unsatisfiable. An upper bound of the time complexity is $O(n^2)$ in the total number of literals, though there exist linear unit propagation algorithms as well –

> either way, it is polynomial.
>
> It's worth seeing what unit propagation "looks like" when we treat a Horn clause $X \vee \neg Y \vee \neg W \vee \neg Z$ as an implication $(Y \wedge W \wedge Z \Rightarrow X)$. A positive unit clause is $\Rightarrow P$: a theorem without any hypotheses. To satisfy this, we of course need to set P to true. Any other assertion of P with hypotheses becomes automatically true (removing all clauses including P), and the hypothesis P can be discharged in any other clause (removing $\neg P$ from every clause). A negative unit clause $\neg P$ corresponds to $P \Rightarrow$, stating that the assumption of P leads to a contradiction – hence, P must be false. When there are no more unit clauses left, every implication will have at least one hypothesis; to satisfy all of them, we set the hypothesis to be false, since falsity implies anything.

3. Reductions

1. We define the complexity class of *quasi-polynomial-time* problems Quasi-P by:

$$\text{Quasi-P} = \bigcup_{k=1}^{\infty} \text{Time}\left(n^{(\log n)^k}\right)$$

Show that if $L_1 \leq_P L_2$ and $L_2 \in$ Quasi-P, then $L_1 \in$ Quasi-P.

> To decide $n \in L_1$, we can apply the reduction to get $f(n)$ and use the decision procedure to determine $f(n) \in L_2$; by the properties of reduction, this will give a decision procedure for L_1: $\chi_{L_1} = \chi_{L_2} \circ f$. We also need to ensure that $L_1 \in$ Quasi-P. The reduction is poly-time, so computing $f(n)$ takes time $O(n^k)$ for some $k \in \mathbb{N}$. Then, we apply the decision procedure for L_2, but with an input of length $f(n)$ – while this can certainly be considerably longer than n, it cannot be longer than the number of time steps it takes to compute, which is at most $c_1 n^k$ for some naturals c and n. Running the decision procedure for L_2 will be an additional $c_2(f(n))^{\log(f(n))^l}$ time, so in total, computing χ_{L_1} takes time
>
> $$\begin{aligned} O\left(x^k\right) + O\left(f(x)^{\log(f(x))^l}\right) &= O\left(x^k + (x^k)^{\log(x^k)^l}\right) \\ &= O\left(x^k + x^{k(k\log x)^l}\right) \\ &= O\left(x^k + x^{k^{l+1}(\log x)^l}\right) \\ &= O\left(x^{(\log x)^l}\right) \end{aligned}$$
>
> which is indeed quasi-polynomial.

2. In general, k-colourability is the problem of deciding, given a graph $G = (V, E)$, whether there is a colouring $\chi : V \to \{1, \ldots, k\}$ of the vertices such that if $(u, v) \in E$, then $\chi(u) \neq \chi(v)$. That is, adjacent vertices do not have the same colour.

   a) Show that there is a polynomial time algorithm for solving 2-colourability.

   > Start by selecting any node, and colouring it one colour (e.g. red). Then, colour every neighbour of this node the second colour (e.g. green), the neighbours of these nodes

red, and so on, alternating between the two colours at each step. If at any point we attept to colour an already coloured neighbour into the opposite colour, the algorithm terminates and returns false – otherwise (all nodes are coloured and there are no conflicts), we succeeded and can return true.

We need to ensure that a conflict exists no matter what the starting node was. Suppose we have an unsuccessful colouring started from a node v, but a successful one when started from a node u. Since the successful colouring started from u must give some colour to u as well, and ensure that no neighbouring nodes have the same colour, a colouring from v cannot ever encounter conflicts – there is no "choice" that can be made differently compared to the colouring from u. Thus, if we find a conflict from one node, we will encounter the conflict from all nodes.

The algorithm is linear in the number of vertices, so $2\text{-colourability} \in \mathrm{P}$.

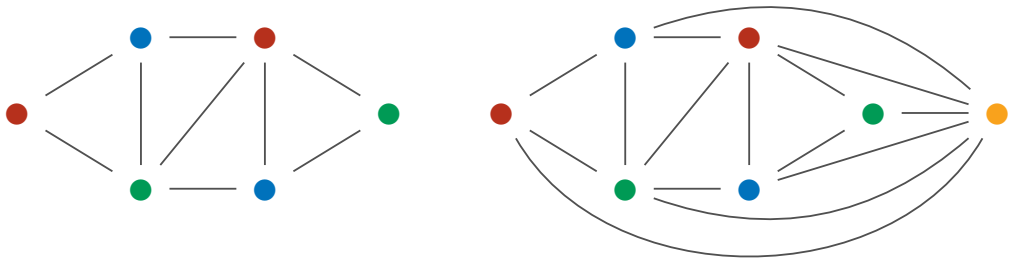
b) Show that, for each k, k-colourability is reducible to $(k+1)$-colourability. Does this, together with part (a), mean that 3-colourability is also in P?

A reduction $r : k\text{-col.} \leq_p (k+1)\text{-col.}$ is a function transforming an input to k-col. to an input of $(k+1)$-col. such that solving the latter gives a solution to the former. That is, if we can solve $(k+1)$-colourability, we can also solve k-colourability by using the decision procedure for $(k+1)$-col. as a "subroutine" or "helper function".

Given any graph $G = (V, E)$, we can determine whether it's k-colourable by adding a new vertex to the graph with edges from all existing vertices to this new vertex, then asking whether the new graph is $k+1$-colourable:

$$(V, E) \in k\text{-col.} \iff (V \uplus \{ v^* \},\ E \uplus \{ (v, v^*) \mid v \in V \}) \in (k+1)\text{-col.}$$

If the new graph G' is $(k+1)$-colourable, any satisfying colouring will have the property that v^* will have a unique colour: since it is connected to every existing vertex in G, none of these can have the same colour as v^*. That is, the original subgraph G of G' will only be coloured with k colours, as the $(k+1)^{\text{th}}$ colour is taken by v^*. Conversely, if G is k-colourable, G' will always have a $(k+1)$-colouring by assigning the new colour to v^*.



It is important to remember that reductions usually go from an "easier" problem

> to a "harder" one – $L_1 \leq_P L_2$ means L_2 is at least as hard to solve as L_1. If we had an expensive algorithm to solve L_1, a cheap algorithm to solve L_2, and a reduction $L_1 \leq_P L_2$, it may be cheaper to solve L_1 by taking a "detour" through L_2. Since in complexity theory we consider the complexity of a whole problem, not just a particular algorithm, $L_1 \leq_P L_2$ means "to solve L_1, we can take a detour through L_2 but it's probably not worth it". We showed that 2-col. is in P, and 2-col. $\leq_P$ 3-col., so 3-col. is "harder" than 2-col.; the reduction doesn't give us a polynomial algorithm for solving 3-col., it just tells us that we can solve 2-col. by taking a potentially expensive detour instead of our polynomial algorithm. In fact, you will later learn that 3-colourability is *much* harder (NP-complete), so taking the detour would be really silly.