Complexity Theory

Solutions

2021

Contents

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 Unary-Prime in the one-letter alphabet $\{ a \}$ defined by Unary-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 Unary-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, Unary-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 Unary-S in the one-letter alphabet $\{ a \}$ defined by Unary-$S = \{ a^n \mid n \in S \}$, and the language Binary-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 Unary-S is in P, then Binary-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.

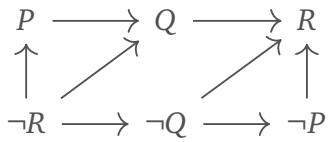
      > Since each clause can be converted to two different implications, every variable will correspond to both a nonnegated and negated graph vertex – thus, for n variables, the graph will contain $2n$ vertices. Similarly, the number of edges will be $2m$, twice the number of clauses. Since some transformations 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. We can rewrite a 3CNF problem as a set of 2CNF clauses, but the transformation results in an exponential increase in the number of clauses.

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 $2\text{-colourability} \in \mathrm{P}$.

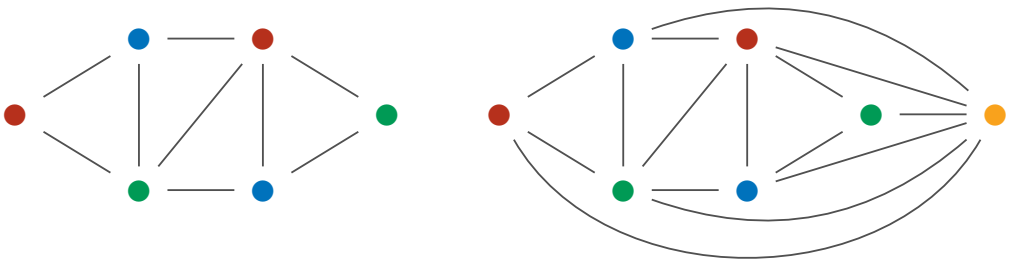
b) Show that, for each k, k-colourability is reducible to $(k+1)$-colourability. Does this, together with part (a), mean that 3-colourability is also in P?

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.

4. NP-completeness

1. Show that the identity function is a poly-time reduction, and composition of poly-time reductions is a poly-time reduction.

   > The Turing machine computing the identity function halts as soon as it is started, since the output of the computation is the input itself. This of course takes constant time, which is polynomial – thus, $L \leq_P L$ for all languages L.
   >
   > Suppose $f : L_1 \leq_P L_2$ and $g : L_2 \leq_P L_3$ are two poly-time reductions. We want to show that $g \circ f : L_1 \leq_P L_3$ is also a poly-time reduction. The Turing-machine computing $g \circ f$ on input x first runs computes $f(x)$ with the machine computing f, then runs the machine computing g on $f(x)$ to get $g(f(x))$. If $x \in L_1$, then $f(x) \in L_2$ (since f is a reduction), and hence $g(f(x)) \in L_3$ (since g is a reduction); similarly if $x \notin L_1$. Thus, $g \circ f$ is a reduction, but we also need to make sure that it is a polynomial-time reduction. Computing $f(x)$ takes time $p_1(x)$ for a polynomial p_1, then continuing with g takes at most $p_2(p_1(x))$ time, since the length of $f(x)$ can be at most polynomial in the length of x. However, polynomials of polynomials are also polynomials (due to the distributivity of multiplication over addition), so $p_2(p_1(x)) = p_3(x)$ for some polynomial p_3. We can therefore conclude that $g \circ f : L_1 \leq_P L_3$ is indeed a poly-time reduction.
   >
   > It's worth noting that this result is not so straightforward for other types of resource-bounded reductions – it makes use of the fact that polynomial functions themselves compose. As a trickier example, consider the composition of logarithmic space reductions – what goes wrong with the "obvious" solution and can it be overcome?

2. A problem in NP is called NP-*intermediate* if it is neither in P nor NP-complete.

   a) Are there any problems that are known to be NP-intermediate?

      > If there were, that would immediately imply that P and NP are distinct, which is not known (though very much suspected).

   b) Research and briefly summarise *Ladner's theorem*.

      > Ladner's theorem states that the converse of the above also holds: if P and NP are distinct, then there exist NP-intermediate languages. This also implies that $\mathrm{P} = \mathrm{NP}$

> if and only if there are no NP-intermediate languages. This is another avenue for trying to tackle $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$: there are several problems that are suspected to be NP-intermediate (such as integer factorisation), because intuitively they seem simpler than other NP-complete problems.

3. Suppose that a language $L_1 \subseteq \Sigma_1^*$ is polynomial-time reducible to a language $L_2 \subseteq \Sigma_2^*$ with the underlying function $f : L_1 \leq_P L_2$. Prove or disprove the following claims, or state if the answer is unknown and explain why:

   a) If $L_2 \leq_P L_1$, then f is a bijection.

   > **False.** As the notes state, the reduction function does not need to be a surjection (i.e. cover its whole codomain): it may map every string in L_1 to a single string in L_2, and all strings outside L_1 to a string outside L_2, and thus have a range of size 2. Taken as a function on strings, f cannot have an inverse. If in addition there is a reduction from L_2 to L_1, its underlying function must be different from f, so $L_2 \leq_P L_1$ does not pose any extra constraints on f.

   b) If f is a bijection, then $L_2 \leq_P L_1$.

   > **Unknown.** If f is a poly-time reduction, asking whether the inverse f^{-1} is also a poly-time reduction (giving us $L_2 \leq_P L_1$) is asking whether one-way functions (see later) exist, which is an unknown problem.

   c) If f is a bijection, then L_2 is in NP.

   > **False.** There are no constraints on L_1 – it could be a language outside of NP. Then, a trivial bijective reduction is the identity reduction $\mathrm{id} : L_1 \leq_P L_1$ (see Ex. 4.1), but $L_2 = L_1$ is not in NP by assumption.

   d) If f is a bijection and L_1 is in NP, then L_2 is in NP.

   > **True.** In fact, it's enough if f is a surjection: that is, for every string $y \in \Sigma_2^*$, there exists a string $x \in \Sigma_1^*$ such that $f(x) = y$. To show that L_2 is in NP, we need to describe a nondeterministic polynomial algorithm to decide whether a $y \in \Sigma_2^*$ is in L_2. Given such an input $y \in \Sigma_2^*$, by surjectivity, there must exist an $x \in \Sigma_1^*$ such that $f(x) = y$. We can find this by nondeterministically guessing an x and polynomially computing $f(x)$ to check whether $f(x) = y$. Once we found such an x, we can use the decision procedure for L_1 to check whether $x \in L_1$, and by the reduction property of f, $f(x) = y$ must be in L_2.

   e) If L_1 is NP-complete, then $L_2 \leq_P L_1$.

   > **False.** Every NP-problem is reducible to L_1, and L_1 is reducible to L_2, so by composition of reductions, every NP-problem is reducible to L_2, making it NP-hard. However, there's no reason for L_2 to be in NP, so it's not necessarily NP-complete – for example, it could be the Halting Problem, which is NP-hard but not in NP.

f) If L_2 is NP-complete, then $L_2 \leq_P L_1$.

> **Unknown.** By the standard result of reductions, if L_2 is in NP then L_1 must also be in NP. To show that $L_2 \leq_P L_1$, we would need to prove that L_1 is NP-complete – however, we don't know that L_1 is NP-hard. But this problem is different from the previous one, since we cannot conclude anything definitively. To show that it's false, we need to exhibit a language L_1 which is in NP but is not in NPC, i.e. an NP-intermediate language – but as shown above, that would immediately imply that $\mathrm{P} \neq \mathrm{NP}$, which we don't know. In fact, by Ladner's theorem, if $\mathrm{P} \neq \mathrm{NP}$, we can construct such a counterexample (making the claim false), but if $\mathrm{P} = \mathrm{NP} = \mathrm{NPC}$, L_2 is in P so it is of course reducible to $L_1 \in \mathrm{P}$. In summary, this statement is equivalent to claiming that $\mathrm{P} = \mathrm{NP}$, which is unknown.

5. NP-complete problems

1. Given a graph $G = (V, E)$, a set $C \subseteq V$ of vertices is called a *vertex cover* of G if, for each edge $(u, v) \in E$, either $u \in C$ or $v \in C$. That is, each edge has at least one end point in C. The decision problem V-COVER is defined as:

   Given a graph $G = (V, E)$, and an integer K, does G contain a vertex cover with K or fewer elements?

   a) Show a polynomial time reduction from IND to V-COVER.

   > Given a graph $G = (V, E)$, $C \subseteq V$ is a vertex cover of G if $V \setminus C$ is an independent set: there are no edges which have either of their endpoints in $V \setminus C$, because at least one endpoint of every edge is in C. Thus, the input $(G = (V, E), K)$ of IND can be transformed to $((V, E), |V| - K)$ as the input to V-COVER, and this poly-time transformation will be a reduction due to the opposite nature of the problems.

   b) Use a) to argue that V-COVER is NP-complete.

   > V-COVER is in NP, since we can verify a candidate vertex cover by analysing the graph and the budget. We also showed that $\mathrm{IND} \leq_P$ V-COVER, and since IND is NP-complete, V-COVER is NP-hard. Put together, we get that V-COVER is also NP-complete.

2. The problem of *four-dimensional matching*, 4DM, is defined analogously with 3DM:

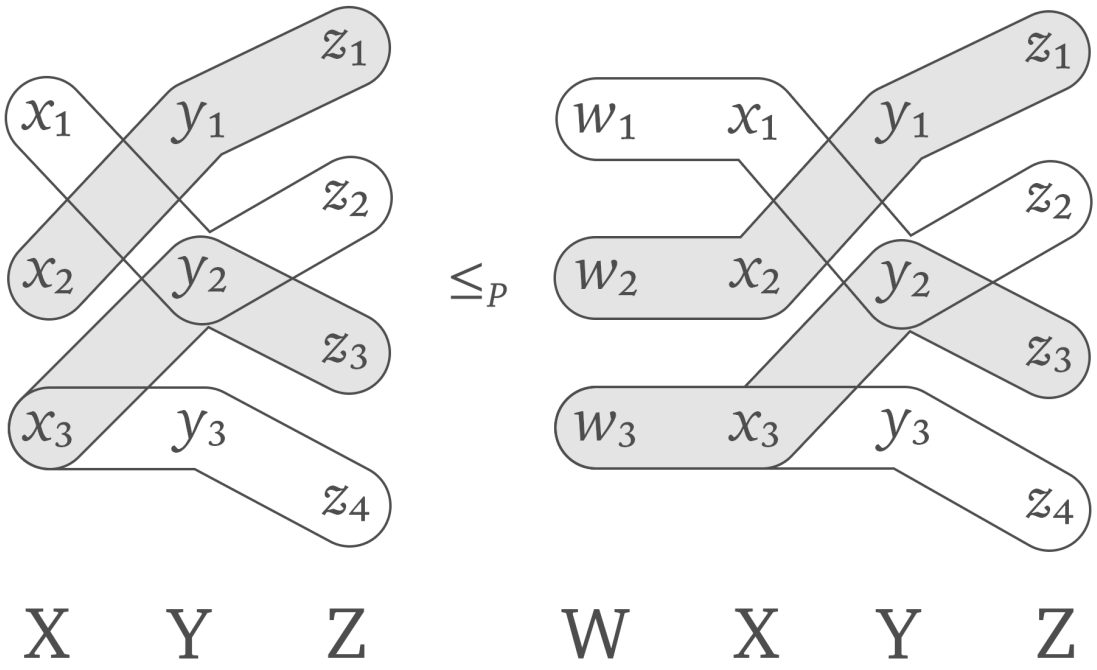
   Given four sets, W, X, Y and Z, each with n elements, and a set of quadruples $M \subseteq W \times X \times Y \times X$, is there a subset $M' \subseteq M$ such that each element of W, X, Y and Z appears in exactly one tuple in M'?

   Show that 4DM is NP-complete.

   > 4DM is in NP since any proposed solution can be checked in polynomial time by seeing if any node in any set appears more than once. To show that 4DM is also NP-hard, we

present a reduction from 3DM (a known NP-complete problem) to 4DM.

If we can find a matching given four sets W, X, Y and Z, any three of those sets (e.g. X, Y, Z) will necessarily have a 3D matching. Hence to transform the input to 3DM into the input to 4DM, we simply add a new set W with the same number of elements as X, and modify every triple (x_i, y_j, z_k) to a 4-tuple (w_i, x_i, y_j, z_k) – that is, we add a one-to-one edge between every pair in W and X, giving us a trivial bipartite matching. These edges cannot affect whether there is a 3D matching for X, Y, and Z, so the original three sets have a 3D matching if and only if W, X, Y, and Z have a 4D matching.



3. Given a graph $G = (V, E)$, a source vertex $s \in V$ and a target vertex $t \in V$, a *Hamiltonian path* from s to t in G is a path that begins at s, ends at t, and visits every vertex in V exactly once. We define the decision problem HamPath as:

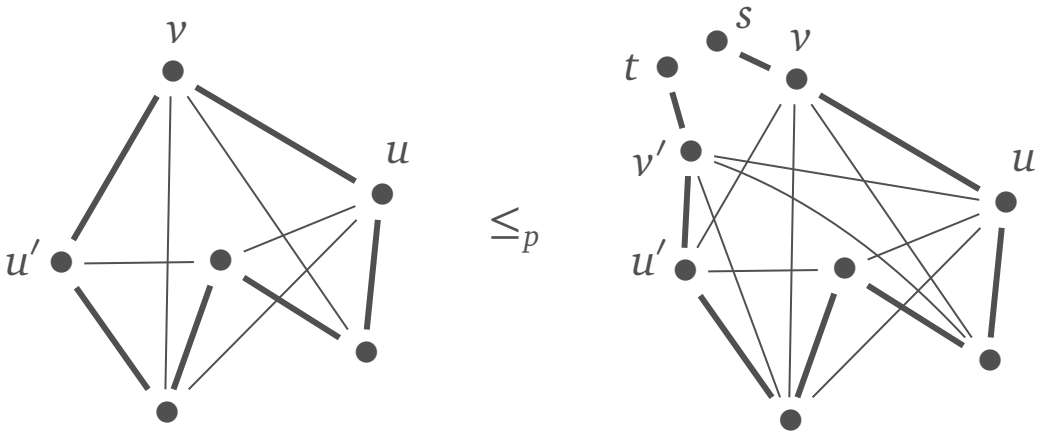
   Given a graph $G = (V, E)$ does G contain a Hamiltonian path?

   a) Give a poly-time reduction from the Hamiltonian cycle problem to HamPath.

   The reduction is not as simple as saying "run the Hamiltonian path algorithm with the same starting point and endpoint", since the endpoints are not inputs to the HamPath problem, and a path with the same endpoints can't be Hamiltonian (since the endpoints would appear twice.). Given that we have no explicit control over the endpoints of the path, we must modify the graph in a way that forces a particular pair of vertices to be the endpoints. We also can't look for a path between any node s and an adjacent node t, because that may not result in a Hamiltonian path even if the original graph has a cycle (and, as always, the reduction property is a bi-implication).

   Take any vertex v of G and make a copy v' which has the exact same connections as v.

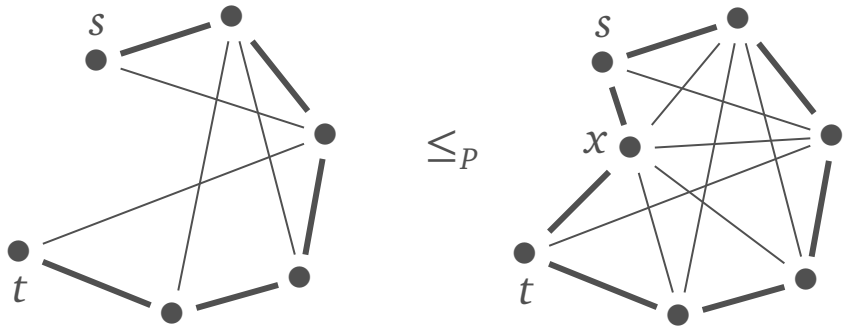
Then, add the nodes s and t, with s connected to v only, and t connected to v' only – call this new graph G'. If G has a Hamiltonian cycle, it must go through v; if we "reroute" the cycle to start from v, then move to v' at the last step instead of returning to v, we can extend this to a path from s to t. If $v \to u \to \ldots \to u' \to v$ is a Hamiltonian cycle in G, the path ($s \to v \to u \to \ldots \to u' \to v' \to t$ will not include any duplicate nodes (as s, t and v' are newly added vertices), so it will also be Hamiltonian. Conversely, if $s \to v \to u \to \ldots \to u' \to v' \to t$ is a Hamiltonian path (and any Hamiltonian path must start at s and end at t, otherwise they would never be reached), by the construction of G' there must be an edge from u' to v (as v' is just a copy of v), and the resulting cycle $v \to u \to \ldots \to u' \to v$ will not contain any duplicate nodes (other than v).



If the graph is directed, this reduction can be simplified to splitting a vertex v into two vertices s and t, with s only containing the outgoing edges of v, and t containing the incoming edges of v. A Hamiltonian cycle in the original graph becomes a Hamiltonian path from s to t, and since the only possible Hamiltonian path in G' must be from s to t (as they can't be intermediate nodes), we can turn one into a cycle by rejoining s and t into v.

b) Give a poly-time reduction from HamPath to the Hamiltonian cycle problem.

Again, we can't just "connect the two ends" of the existing Hamiltonian path, since we don't know what the endpoints are. However, we can do something quite dramatic which will definitely connect the endpoints of a path, if it exists: add a new node x to the graph, and connect it to every other node with a single edge. Now, if G has a Hamiltonian path $s \to v \to \ldots \to v' \to t$ between s and t, this can be extended in G' to $s \to v \to \ldots \to v' \to t \to x \to s$ into a cycle – and as the only new node we added to the cycle is x, it must be Hamiltonian. Conversely, if we can find a Hamiltonian cycle in G', it must go through x preceded by some node t' and succeeded by some node s'; if we remove x and all its connections, we will be left with a Hamiltonian path from s' to t', as it will be a subpath of the original cycle.



c) Consider the following, modified statement of the Hamiltonian path problem:

Given a graph $G = (V, E)$ and vertices $s, t \in V$,
does G contain a Hamiltonian path from s to t?

Explain how this differs from the problem above, and comment on whether your reductions in parts a) and b) can be simplified for this version.

> The difference is that the endpoints s and t are now inputs to the problem: we are asking whether there exists a Hamiltonian path between a specific pair of vertices. This of course means that even if a graph has a Hamiltonian path, there might not be one between two particular points.
>
> Knowing the endpoints lets us simplify the more general algorithms above. In the reduction of HamPath to HAM, we needed to add a new node connect it to every other node because we didn't know where the Hamiltonian path started. In this case, s and t are part of the input, so it is enough to connect the new node to s and t only. Why can't we just add an edge between s and t directly? While one direction of the reduction proof would work (if there is a path between s and t, it can be closed into a cycle with one edge), there is no guarantee that a Hamiltonian cycle in the modified graph would go through our new edge – there's no reason s and t would need to be adjacent. By explicitly adding a new node we force the cycle to have s and t next to each other (separated by x), which can be turned into a path by removing x.
>
> To reduce HAM to HamPath, we still can't "call" the path algorithm with the same start and endpoint (since one node can't appear twice in the path), or with an arbitrary node and one of its neighbours (the edge connecting them may not be part of a Hamiltonian cycle, of which there may only be one in the graph). We have to create a copy v' of one of the nodes v, but we need not add explicit source and target vertices s and t, since v and v' can be made the inputs of the modified path problem.

4. We know from the Cook–Levin Theorem that every problem in NP is reducible to SAT. The proof worked for a general nondeterministic Turing-machine, but for some problems it is easy to give an explicit reduction. Describe how to obtain, for any graph $G = (V, E)$, a Boolean expression φ_G such that φ_G is satisfiable if and only if:

   a) G is 3-colourable.

The search space for possible solutions to graph-colourability is colour assignments for every vertex of a graph. If the problem was 2-colourability, a Boolean variable could correspond to one colour. This is not the case for three colours, so we need a way to encode colour as two or more Boolean variables. For simplicity, we can use "one-hot" encoding, with three Boolean variables R, G and B of which only one can be true at a time. We have three such variables R_i, G_i and B_i for every graph node $v_i \in V$, for a total of $3|V|$ variables. To encode a potential instance to the problem, i.e. a specific assignment of colours to vertices, we need to ensure that every vertex has at most one colour:

$$\bigwedge_{v_i \in V} (R_i \wedge \neg G_i \wedge \neg B_i) \vee (\neg R_i \wedge G_i \wedge \neg B_i) \vee (\neg R_i \wedge \neg G_i \wedge B_i) \qquad ①$$

Finally, to determine whether an assignment of colours is a valid 3-colouring, we require that no two adjacent nodes have the same colour:

$$\bigwedge_{(v_i, v_j) \in E} \neg(R_i \wedge R_j) \wedge \neg(G_i \wedge G_j) \wedge \neg(B_i \wedge B_j) \qquad ②$$

Conjoining these two expressions will result in a Boolean formula which is satisfiable if and only if the graph is 3-colourable: on the one hand, ① every node has a unique colouring and ② no adjacent nodes have the same colour; on the other, if the graph has a 3-colouring, the three colour variables corresponding to the vertex colours can be uniquely set to satisfy the formula.

b) G contains a Hamiltonian cycle.

The search space is the set of possible vertex orderings. A simple way to encode this is with $|V|^2$ variables $H_{p,v}$ which indicate whether the vertex v is in position p along the Hamiltonian cycle. The collection of variables represents a path of length $n = |V|$ in the graph if:

- every slot in the path is occupied, i.e. for every position p, one of the $H_{p,v}$ must be true:
$$\bigwedge_{p=1}^{n} \bigvee_{v \in V} H_{p,v}$$
- two adjacent vertices in the path must correspond to adjacent vertices in the graph, i.e. for every pair of consecutive positions there must exist a corresponding edge in the graph:
$$\bigwedge_{p=1}^{n-1} \bigvee_{(v,u) \in E} H_{p,v} \wedge H_{p+1,u}$$

A path is Hamiltonian if:

- every vertex appears in the path, i.e. for every vertex v, one of the $H_{p,v}$ must be

true:

$$\bigwedge_{v \in V} \bigvee_{p}^{n} H_{p,v}$$

- a vertex appears only once in the path, i.e. if a vertex is at two positions, the positions must be equal:

$$\bigwedge_{v \in V} \bigwedge_{p=1}^{n} \bigwedge_{q=1}^{n} (H_{p,v} \wedge H_{q,v}) \Longrightarrow p = q$$

Finally, a Hamiltonian path can be closed in a cycle if there is an edge from the last node in the path to the first:

$$\bigvee_{(v,u) \in E} H_{n,v} \wedge H_{1,u}$$

Taking the conjunction of these five conditions, we ensure that any satisfying variable assignment can be directly mapped into a Hamiltonian cycle (just reading off the position and vertex for every variable set to true), and any Hamiltonian cycle in the graph will give a satisfying interpretation by setting the relevant variables to true.

An alternative collection of constraints is as follows, where $P = [1, |V|)$ be the set of position:

- every vertex is in at least one position

$$\bigwedge_{v \in V} \bigvee_{p \in P} H_{p,v}$$

- if a vertex v is at position p, then it cannot be at any other position $q \neq p$ *and* no other vertex $u \neq v$ can be at that position; i.e. every vertex must be in *at most* one position:

$$\bigwedge_{v \in V} \bigwedge_{p \in P} H_{p,v} \Longrightarrow \bigwedge_{q \in P \setminus \{p\}} \neg H_{q,v} \wedge \bigwedge_{u \in V \setminus \{v\}} \neg H_{p,u}$$

- every adjacent pair of nodes must have an edge connecting them (note the modular arithmetic to complete the cycle):

$$\bigwedge_{p \in P} \bigvee_{(u,v) \in E} H_{p,u} \wedge H_{p+1 \pmod{|V|},v}$$

Hint: By analysing the search space of the problem, determine a collection of Boolean variables that can encode the relevant properties of the graph (cf. $S_{i,q}$, $T_{i,j,\sigma}$ and $H_{i,j}$ in the Cook–Levin Theorem proof). Give the constraints on the variables which are required to make the encoded graph an instance of the given problem (cf. expressions (1)-(7) in the CLT proof). Combine these with the constraints that would decide whether a potential instance is a member of the problem or not (cf. expression (8) in the CLT proof).

5. An instance of a *linear programming* problem consists of a set $X = \{x_1, \ldots, x_n\}$ of variables and a set of constraints, each of the form

$$\sum_{1 \leq i \leq n} c_i x_i \leq b,$$

where each c_i and b is an integer.

The 0-1 Integer Linear Programming Feasibility problem 01-ILP is defined as follows:

> *Given an instance of a linear programming problem, determine whether there is an assignment of values from the set $\{0, 1\}$ to the variables in X so that substituting these values in the constraints leads to all constraints being simultaneously satisfied.*

Prove that this problem is NP-complete.

As usual, there are two things to prove: that the problem is in NP, and that it is NP-hard. The first part is easy: given a proposed assignment of the variables, we can check whether all constraints are satisfied in $O(cn)$ time, where n is the number of variables and c is the number of constraints. NP-hardness is established by reducing another NP-complete problem to 01-ILP. Many NPC-problems can be naturally expressed using integer constraints, so there are many possible reductions – three examples are below. Note that a greater-than constraint $c_i x_i \geq b$ can be expressed in the appropriate form as $-c_i x_i \leq -b$.

CNF $\leq_P$ 01-ILP. For each propositional variable P in the clauses we create a 01-ILP variable x_P which equals 1 or 0 depending on whether P is true or false, respectively. Each clause is mapped to a constraint on its variables, expressing that at least one of them must be 1. This is achieved by adding together x_P for a literal P, and $(1 - x_P)$ for a literal $\neg P$. For example, the clause $\{P, \neg Q, R\}$ becomes the constraint $x_P + (1 - x_Q) + x_R \geq 1$. By construction, any satisfying assignment of the clauses will satisfy the constraints, since at least one of the terms in the sum will be 1. Conversely, an assignment to the variables will give an interpretation as the constraints force at least one literal to be true in every clause. Put together, the explicit reduction mapping is

V-COVER $\leq_P$ 01-ILP. Recall the vertex cover problem: given a graph $G = (V, E)$, and a budget K, is there a set $C \subseteq V$ of vertices with K or fewer elements such that at least one endpoint of every edge of G is in C? The reduction of an instance (G, K) to 01-ILP is as follows. For every vertex $v \in V$, we have a variable x_v whose truth value encodes whether the v is in the vertex cover or not. The main property of a vertex cover is that includes at least one endpoint for every edge; we can capture this as a numeric constraint of the variables by requiring that for every edge $(u, v) \in E$, the constraint $x_u + x_v \geq 1$ holds. Now, any assignment satisfying these constraints will be a vertex cover, and to express the budget constraint, we also add $\sum_{v \in V} x_v \leq K$ to make the number of variables set to true sufficiently small. If G has a vertex cover smaller than K, the constraints will be satisfied by construction, and from satisfying assignment to the variables we can read off the variables which should be included in the vertex cover.

SUBSET-SUM $\leq_P$ 01-ILP. Recall the subset sum problem: given a set S of numbers and a target t, is there a subset of the numbers which adds up exactly to t? Given an instance (S, t) of SUBSET-SUM, we associate every $n \in S$ with a variable x_n. The subset sum

> condition is satisfied if the sum $\sum_{n \in S} n \cdot x_n$ is equal to t; as less-than-or-equal constraints, this is simply expressible as the conjunction of $\sum_{n \in S} n \cdot x_n \leq t$ and $\sum_{n \in S} -n \cdot x_n \leq -t$.

6. *Self-reducibility* refers to the property of some problems in $L \in \mathrm{NP}$, where the problem of finding a witness for the membership of an input x in L can be reduced to the decision problem for L. This question asks you to give such arguments in three specific instances.

   a) Show that, given an oracle (i.e. a black box) for deciding whether a formula φ over a set of variables $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is satisfiable, there is a polynomial-time algorithm that gives a variable assignment which satisfies a formula over $\mathcal{V}$.

   > The idea behind self-reducibility is that we perform repeated calls to the decision procedure which restrict the space of possible solutions after each call. In the case of SAT, each call to the oracle reduces the search space (all possible assignments) by half, so we can get a satisfying assignment to all n variables in $n + 1$ steps. In the first step, we check if φ is satisfiable – if it isn't, we can't find a satisfying assignment anyway. Once we know that there is a suitable assignment, we call the oracle with a derived formula $[\top/x_1]\varphi$, in which the variable x_1 is assigned to be $\top$. If the oracle says that the formula is still satisfiable, we know that there must be an interpretation with x_1 set to true, but if it isn't, x_1 must be set to false. Based on the answer, we can get a partially assigned satisfiable φ_1, which is φ with its variable x_1 set to the appropriate value. We now do the same with $[\top/x_2]\varphi_1$ to get φ_2 and so on, with φ_{k+1} defined as $[\top/x_k]\varphi_k$ or $[\bot/x_k]\varphi_k$ depending on which one is satisfiable. In the end, we will end up with a satisfying assignment to all n variables of the formula.
   >
   > Note that the partial assignments need to be accumulated, and it is not enough to "sample" $[\top/x_k]\varphi$ for all k independently. This is because the interpretations that the oracle discovers may be different: if $[\top/x_1]\varphi$ and $[\top/x_2]\varphi$ are both satisfiable, the two assignments may set different values to the other variable so $[\top/x_1, \top/x_2]\varphi$ may become unsatisfiable. As an example of this, consider the formula $x_1 \vee (\neg x_1 \wedge x_2)$.

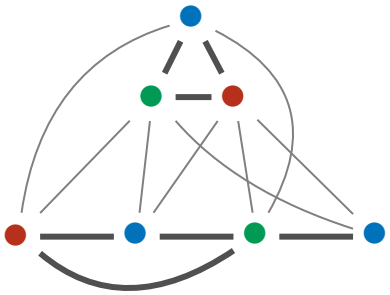
   b) Show that, given an oracle for deciding whether a given graph $G = (V, E)$ is Hamiltonian, there is a polynomial-time algorithm that, on input G, outputs a Hamiltonian cycle in G if one exists.

   > A graph is Hamiltonian if it contains a Hamiltonian cycle. As with SAT, a graph may have several Hamiltonian cycles, but we can perform repeated calls to the oracle to zero in on a particular one. The process is easy: if the initial graph is Hamiltonian, we repeatedly remove edges of the graph and call the oracle until the graph is no longer Hamiltonian; at that point, the last edge we removed must be part of a cycle. We return and fix this edge, and continue with the rest of the graph until the cycle is found. The most number of calls made to the polynomial oracle is $O(|E|) \leq O(n^2)$, so the whole process is still polynomial.

   c) (Harder) Show that, given an oracle for deciding whether a given graph G is 3-colourable, there is a polynomial-type algorithm that, on input G, produces a valid 3-colouring of G if

one exists.

> The difficulty with graph colourability is that our previous trick of partially assigning or modifying the input does not work: we can't call the oracle with a partial colouring, as that is not part of the input and not an inherent property of the graph (like the edges were in the Hamiltonian graph problem). In addition, "knowing" that a particular vertex is red doesn't tell us anything, since the colours themselves do not matter.
>
> One approach is figuring out which nodes have the same colour. Given a graph G, we of course first make sure that it's 3-colourable. If G has more than three vertices, there must be at least two unconnected nodes of the same colour. Pick two unconnected vertices u and v and construct the graph G' in which u and v are merged: that is, u and v are replaced with a new node w, and every edge connected to either u or v now connects to w. If there is a 3-colouring of G in which u and v have the same colour (e.g. red), all of their neighbours must be green or blue. In that case, there is a colouring of G' in which w is red and all of its neighbours are either green or blue. That is, G' must still be 3-colourable, so there is a colouring of G which assigns the same colour to u and v. If G' is not 3-colourable anymore, there is no solution that assigns the same colour to u and v; we backtrack and pick two different nodes. This process of merging unconnected nodes and checking whether the graph stays 3-colourable is repeated until we end up with a complete 3-vertex graph (a triangle). In the end, we can partition the nodes of G based on which of the final three triangle nodes they "contributed to", which will determine a valid 3-colouring of the graph.
>
> Another approach is introducing a triangle to the graph, initially unconnected to G. If G is 3-colourable, G' constructed by adding the triangle to G must also be 3-colourable, since the three new nodes can have the three different colours. For concreteness, we can even label the new nodes with the colours they are intended to represent (but remember, colourability is really a vertex partitioning problem, the colours are just an intuitive abstraction). We then use the triangle to "sample" the colours of vertices from G as follows: if G has a colouring which assigns a vertex v the colour red, connecting it to the green and blue vertices of the triangle will maintain the 3-colourability of G'. We repeat this, guessing a colour for each vertex by connecting it to the two different-coloured nodes of the triangle and checking if the new graph is 3-colourable. If it isn't, we backtrack and guess another colour. Eventually we end up with G' with extra edges connecting the original vertices of G to exactly two vertices in the triangle. The output colour for each vertex is the colour of the triangle node it is *not* connected to.



Optional exercises

1. The problem E3SAT is defined as follows:

   > *Given a set of clauses, each clause being a disjunction of exactly three distinct literals and containing exactly three distinct variables, determine whether it is satisfiable.*

   Prove that E3SAT is NP-complete. *Hint:* introduce new variables to the set by adding a tautological clause.

   > A simple initial attempt could be a reduction from 3SAT that duplicates one or more literals in one or two-literal clauses. However, the distinctness requirement makes this unsuitable – we need to introduce new variables into the set. It's not enough to arbitrarily "pad" short clauses with new variables since we may end up satisfying an initially unsatisfiable set of clauses: for instance $\{ P \}\ \{ \neg P \}$ is not satisfiable, but $\{ P, A, B \}\ \{ \neg P, C, D \}$ is.
   >
   > The trick is to add the new variables in a new tautological clause that does not affect the satisfiability of the clause set, namely $\{A, \neg A\}$. Then, we can observe that any literal L can be combined with the new clause using $L \wedge (A \vee \neg A) \simeq (L \vee A) \wedge (L \vee \neg A)$, giving two new clauses $\{ L, A \}, \{ L, \neg A \}$ which are logically equivalent to L. If the literal L is satisfiable, the same interpretation will satisfy the extended clauses (with A set to an arbitrary truth value). If the new pair of clauses is satisfiable, it must assign a truth value to A. Whatever the assignment, we end up with the clauses $\{ L, \top \}, \{ L, \bot \}$, of which the first one is automatically true, while the second one is equivalent to L itself.
   >
   > This technique can be used to soundly increase the size of any clause with an extra literal, which is precisely what we need to do with the 1- and 2-literal clauses in the set. In fact, this even works with empty clauses – while they obviously imply a contradiction right away, the reduction has to handle them uniformly. If the original 3SAT instance is satisfiable, the same assignment will satisfy the reduced E3SAT instance, since the old clauses are subclauses of the extended ones. Conversely, if the extended clause set is satisfiable, it must assign truth values for the newly introduced variables which will simplify the problem to the original clause set.

2. We use $x; 0^n$ to denote the string that is obtained by concatenating the string x with a separator ; followed by n occurrences of 0. If $[M]$ represents the string encoding of a non-deterministic

Turing machine M, show that the following language is NP-complete:

$$S = \{ [M]; x; 0^n \mid M \text{ accepts } x \text{ in } n \text{ steps} \}$$

Hint: Rather than attempting a reduction from a particular NP-complete problem, it is easier to show this from first principles, i.e. construct a reduction for any NDTM M and polynomial bound p.

> S is in NP: given a string $c; x; 0^n$, we split it up along the semicolons, decode c into a TM M, simulate M on the input x, and count the number of steps it takes and checking if that equals n. Every step of this process – while complicated – can be done in polynomial time, so this is a poly-time verifier, as required.
>
> We can show that S is NP-hard from first principles, by exhibiting a poly-time reduction for any language L in NP to S. If $L \in \text{NP}$, there exists a NDTM M that accepts it in polynomial time. The reduction $L \leq_P S$ maps an input string x to $f(x) = [M]; x; 0^n$ as follows:
>
> 1. Encode M to get $[M]$, then write $[M]; x;$ onto the tape. Both steps can be done in polynomial time, since the machine M has a finite number of states and symbols, as well as a finite transition table.
> 2. Simulate M on the input x, appending a 0 to the output at every computation step. Since M is assumed to operate in polynomial time, the number of 0-s appended – and hence the number of steps this stage of the reduction takes – will also be polynomial.
>
> We end up with the string $[M]; x; 0^n$, and by the definition of S, this is in S exactly when M accepts x in n steps, which is how the string was constructed to begin with.