Complexity Theory

Supervision 2 – Solutions

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. It may be true if $\mathrm{P} = \mathrm{NP}$ because every language in P is poly-time reducible to any other language in P (the reduction can simply decide the second language). It may also be false if $\mathrm{P} \neq \mathrm{NP}$ and L_1 is in P or is NP-intermediate.

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 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:

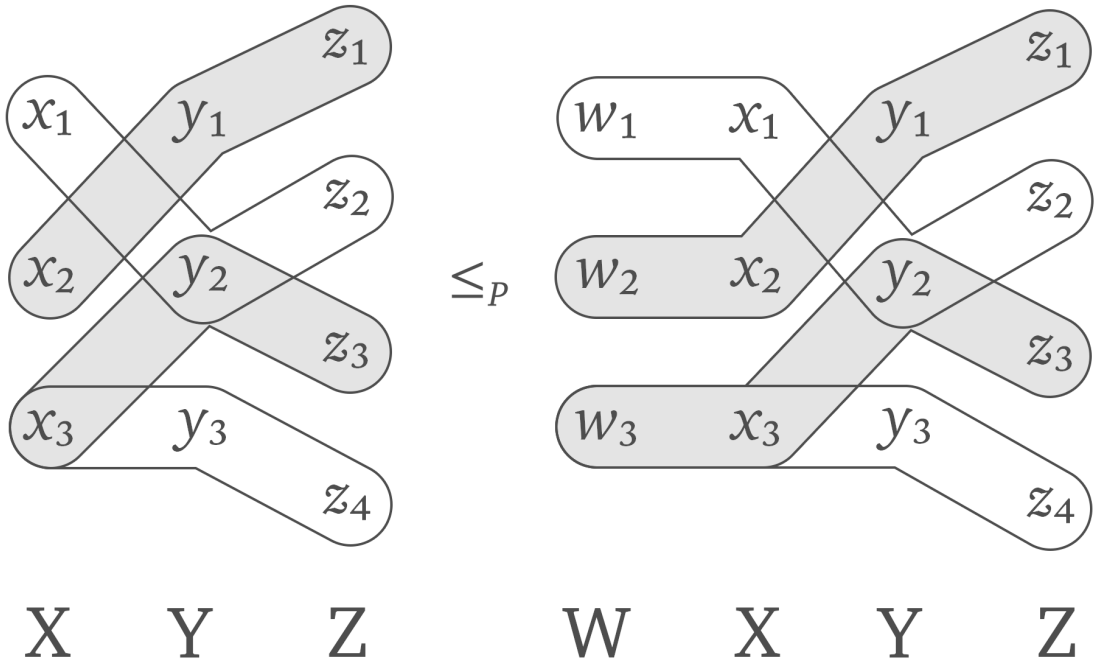
   Given four sets, W, X, Y and Z, each with n elements, and a set of quadruples $M \subseteq W \times X \times Y \times Z$, is there a subset $M' \subseteq M$ such that each element of W, X, Y and Z appears in exactly one tuple in M'?

   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. (Note that the matching

problem can be generalised to sets of different size, as in the diagram below.)



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:

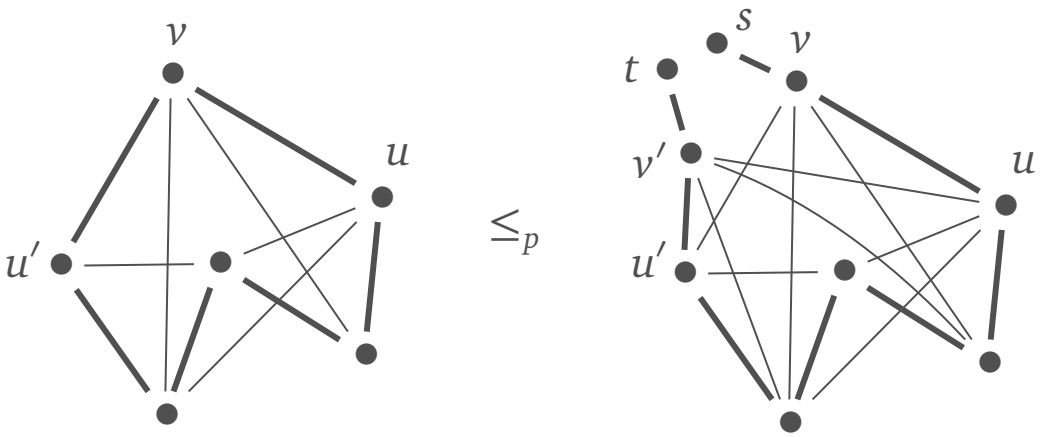
   Given a graph $G = (V, E)$ does G contain a Hamiltonian path?

   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

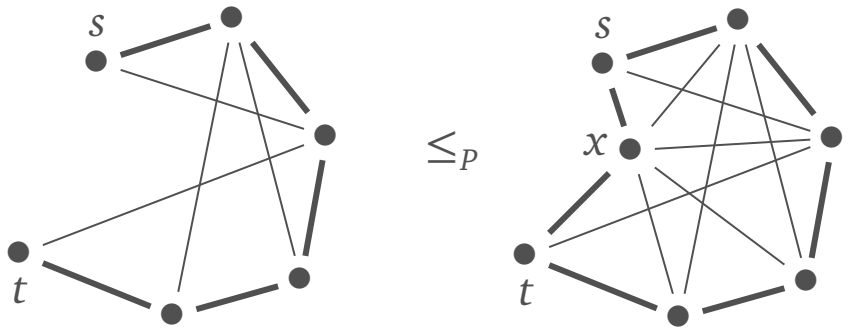
(and any Hamiltonian path must start at s and end at t, otherwise they would never be reached), by the construction of G' there must be an edge from u' to v (as v' is just a copy of v), and the resulting cycle $v \to u \to \ldots \to u' \to v$ will not contain any duplicate nodes (other than v).



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 $\mathrm{HamPath}$ 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 $\mathrm{HamPath}$, 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:

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

- 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 $L \in \text{NP}$, 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.

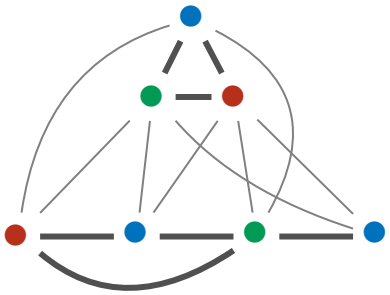
> A graph is Hamiltonian if it contains a Hamiltonian cycle. As with SAT, a graph may have several Hamiltonian cycles, but we can perform repeated calls to the oracle to zero in on a particular one. The process is easy: if the initial graph is Hamiltonian, we repeatedly remove edges of the graph and 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.