Section 9: Solutions

1. SATisfy This

Determine whether each instance of 3SAT is satisfiable. If it is, list a satisfying variable assignment.

(a) $(\neg a \vee \neg b \vee c) \wedge (a \vee c \vee \neg d) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee c \vee \neg d)$
Solution:

This is satisfiable: $a = T, b = F, c = T, d = F$ makes each clause true, so the overall formula is true.

(b) $(\neg a \vee b \vee d) \wedge (\neg b \vee c \vee d) \wedge (a \vee \neg c \vee d) \wedge (a \vee \neg b \vee \neg d) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee \neg d) \wedge (a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$
Solution:

This is unsatisfiable.

(c) $(a \vee \neg c \vee d) \wedge (\neg a \vee b \vee c) \wedge (b \vee \neg c \vee \neg d) \wedge (a \vee \neg c \vee d) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg d) \wedge (\neg a \vee c \vee d) \wedge (b \vee \neg c \vee d) \wedge (a \vee c \vee \neg d)$
Solution:

This is satisfiable: $a = F, b = T, c = T, d = T$ makes each clause true, so the overall formula is true.

(d) $(\neg a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c)$
Solution:

This is unsatisfiable.

2. A Fun Reduction

Define 5SAT as the following problem:
Input: An expression in CNF form, where every term has exactly 5 literals.
Output: `true` if there is a variable setting which makes the whole expression true, `false` otherwise.

And 3SAT as in class:
Input: expression in CNF form, where every term has exactly 3 literals.
Output: `true` if there is a variable setting which makes the whole expression true, `false` otherwise.

Prove that 5SAT is NP-complete using 3SAT.

2.1. Read and Understand the Problem

Read the problem and answer these quick-check-questions.

Make sure you understand 5SAT.

- What is the input type?
- What is the output type?
- Are any words in the problem technical terms? Do you know them all?

Solution:

For 5SAT

- input: an expression in CNF form of n Boolean variables where each clause has 5 literals
- output: true or false (depending on if we have a variable setting which makes the whole expression true)
- CNF form is AND of ORs like $(z_a \vee z_b \vee z_i) \wedge (z_c \vee z_i \vee z_j) \wedge ...$, literals z_i are Boolean variables or the negation of Boolean variables x_i or $\neg x_i$

You're going to design a reduction – what will that reduction look like?

- Which problem are you solving, and which problem are you assuming you have an algorithm for? Make sure your reduction is "going the right direction"
- What is the output type for your reduction?

Solution:

For the reduction

- We want to show that 5SAT is NP-complete, so we need to reduce 3SAT, an NP-complete problem, to 5SAT in polynomial time. We can assume we have an algorithm for 5SAT. In other words, we want to show that $3\text{SAT} \leq 5\text{SAT}$
- output of the reduction: a Boolean which is the answer to the 3SAT (which we get by calling 5-SAT like a library function)

2.2. Design the Reduction

Now write a reduction. Remember a reduction is an algorithm! It often helps to think about the "certificates" (the thing that makes it a YES instance), and transform from one type of certificate to the other. **Solution:**

Let $x_1, \ldots x_n$ be the variables in the 3SAT instance and $C_1, C_2, \ldots, C_m$ be the clauses.

Create two dummy variables d_1, d_2. For each clause C_i, create four clauses:
$C_1 \vee d_1 \vee d_2$
$C_1 \vee \neg d_1 \vee d_2$
$C_1 \vee d_1 \vee \neg d_2$
$C_1 \vee \neg d_1 \vee \neg d_2$

Our 5-SAT instance is: $x_n \ldots, x_n, d_1, d_2$
The $4m$ clauses described above.

2.3. Write The Proof

- To be NP-Complete, 5SAT needs to be in NP. Argue that it is (this argument is usually only 2-3 sentences).
  (a) What is the certificate?
  (b) How does a verifier check it efficiently? **Solution:**

  A verifier would take in the settings of the variables to true and false. Given a setting, a verifier would check that each clause (i.e., each constraint) is satisfied. This will take time linear in the length of the constraints, so it is polynomial time.

- Show your reduction is correct. Remember you need to prove two implications **and** that the running time is polynomial.

(a) Running time: **Solution:**

Running Time: Our algorithm makes 4 copies of every clause and adds a constant length set of literals to each clause, so the running time to create the instance is polynomial (and we call the library only once, which is also at most polynomial).

(b) Correctness: **Solution:**

Correctness
Let ϕ_3 be our 3SAT instance and ϕ_5 be our 5SAT instance.

Suppose ϕ_3 is satisfiable, we show that our reduction returns true. Since ϕ_3 is satisfiable, there is a setting of the variables which causes ϕ_3 to be true. Take that setting, and set d_1, d_2 arbitrarily. Every clause of ϕ_5 is a clause of ϕ_3 with extra literals ORed on, so since each clause of ϕ_3 is true, each clause of ϕ_5 is as well, and this is a satisfying assignment.

Conversely, suppose that our reduction returns true, and therefore ϕ_5 was satisfiable. Consider a satisfying assignment for ϕ_5. We claim that (ignoring d_1, d_2) the same assignment satisfies ϕ_3. Consider an arbitrary clause C_i of ϕ_3. In ϕ_5 there were four clauses built from C_i (each ORed with all combinations of literals of d_1, d_2. One of the created clauses in ϕ_5 had both inserted literals involving d_1, d_2 being false (since we included all possible combinations). Since ϕ_5 was satisfied, this clause evaluated to true, which means that C_i evaluated to true. Since C_i was arbitrary, we have that every clause is true, and therefore a satisfying assignment for ϕ_3, as required.

3. A Reduction between different kinds of problems

Define integer linear programming (ILP) as follows:
Input: An integer matrix A and integer vector b
Output: `true` if there is an integer vector x such that $Ax \leq b$, `false` otherwise.

And 3SAT as earlier:
Input: expression in CNF form, where every term has exactly 3 literals on different variables.
Output: `true` if there is a variable setting which makes the whole expression true, `false` otherwise.

We already know from class that 3SAT $\leq_P$ ILP by a long series of reductions. Prove this directly using a single reduction.

3.1. Read and Understand the Problem

Read the problem and answer these quick-check-questions.

Make sure you understand ILP and 3SAT.

- What is the input type?
- What is the output type?
- Are any words in the problem technical terms? Do you know them all?

Solution:

For ILP

- input: an integer matrix and an integer vector
- output: true or false
- No

You're going to design a reduction – what will that reduction look like?

- Which problem are you solving, and which problem are you assuming you have an algorithm for? Make sure your reduction is "going the right direction"
- What is the output type for your reduction?

Solution:

For the reduction

- We want to show that 3SAT $\leq_P$ ILP. Assume we have an algorithm for ILP. We're trying to solve 3SAT.
- output of the reduction: a boolean which is the answer to the 3SAT (which we get by calling ILP like a library function)

3.2. Design the Reduction

Now write a reduction. Remember a reduction is an algorithm! It often helps to think about the "certificates" (the thing that makes it a YES instance), and transform from one type of certificate to the other.

Solution:

With 3SAT, examples are always much clearer to think about down than the general form. So here's an example (you wouldn't actually write this in your solution):

$$(\neg w \vee \neg x \vee y) \wedge (w \vee y \vee \neg z) \wedge (w \vee \neg y \vee \neg z) \wedge (w \vee x \vee y)$$

We need to use the ILP to handle two things: The Boolean part of 3SAT and the clause constraints.

Let's start with the Boolean part. It seems natural to have the ILP have a variable for each variable for 3SAT. To make sure that these variables are Boolean, we add the constraints:

$$w \leq 1, \qquad x \leq 1, \qquad y \leq 1, \qquad z \leq 1.$$

And as usual with standard form, we always have the constraints

$$w \geq 0, \qquad x \geq 0, \qquad y \geq 0, \qquad z \geq 0.$$

We now need the clause part:

- To represent the value of the negation of w, we can write $1 - w$.
- To represent $\neg w \vee \neg x \vee y$, we want to saw that at least one of the literals is true, so we add the constraint $(1 - w) + (1 - x) + y \geq 1$.
- We can rearrange this to standard form as $w + x - y \leq 1$.

Similarly for the other clauses, we get:

- $w + y + (1 - z) \geq 1 \longrightarrow -w - y + z \leq 0$
- $w + (1 - y) + (1 - z) \geq 1 \longrightarrow -w + y + z \leq 1$
- $w + x + y \geq 1 \longrightarrow -w - x - y \leq -1$

We now write the general form. Let $x_1, \ldots, x_n$ be the variables of the 3SAT instance and $C_1, C_2, \ldots, C_m$ be the clauses.

We add the constraints:

(a) $x_i \leq 1$ for all $i = 1, \ldots, n$

(b) $x_i \geq 0$ for all $i = 1, \ldots, n$, as always

(c) For each clause C_j, if the variables are $x_{i_1}, x_{i_2}, x_{i_3}$, include the constraint:

$$\sum_{k=1,2,3} \begin{cases} x_{i_k} & \text{if the literal } x_{i_k} \text{ appears in } C_j \\ 1 - x_{i_k} & \text{if the literal } \neg x_{i_k} \text{ appears in } C_j \end{cases} \geq 1,$$

which is equivalent to

$$\sum_{k=1,2,3} \begin{cases} -x_{i_k} & \text{if the literal } x_{i_k} \text{ appears in } C_j \\ x_{i_k} & \text{if the literal } \neg x_{i_k} \text{ appears in } C_j \end{cases} \leq -1 + \text{number of negative literals in } C_j$$

in standard form.

3.3. Write The Proof

(a) To be NP-Complete, ILP needs to be in NP. Argue that it is (this argument is usually only 2-3 sentences).

Solution:

The certificate is the integer vector x that is a supposed solution. We need that the integer vector can be assumed to have values that aren't too big or it wouldn't be a short certificate. This is true because of what we already showed for LP. (Any LP solution only needs a polynomial number of bits and any ILP solution satisfies the LP constraints so it does also.)

The verifier would check that $Ax \leq b$. If A has m rows (m inequalities) and n columns (n variables), this takes $O(mn)$ time, so it is polynomial time.

(b) Show your reduction is correct. Remember you need to prove two implications **and** that the running time is polynomial.

Solution:

Running Time: The reduction creates one equation for every clause, so it is definitely polynomial time.

Correctness
We need to show that the correct answer is true if and only if our reduction returns true.

($\implies$) Let φ be a 3-SAT instance and suppose it is satisfiable. We need to show that the system of inequalities we made has a solution. To show this, we should construct x that satisfies the system of inequalities.

Because φ is satisfiable, consider a satisfying assignment. If x_i is assigned true, then in our construction let $x_i = 1$. If x_i is assigned false, let $x_i = 0$.

This assignment clearly satisfies inequalities of the form $x_i \leq 1$ and $x_i \geq 0$. It also satisfies our constraints of the form

$$\sum_{k=1,2,3} \begin{cases} x_{i_k} & \text{if the literal } x_{i_k} \text{ appears in } C_j \\ 1 - x_{i_k} & \text{if the literal } \neg x_{i_k} \text{ appears in } C_j \end{cases} \geq 1,$$

because each clause has at least one literal true, which by our construction means at least one of the summands is 1, and the remaining summands are at least 0, so their sum is at least 1.

($\impliedby$) Suppose that the ILP obtained from converting a 3SAT formula φ returns true, and we need to show that φ is satisfiable. To show this, we need to construct an assignment to the variables of φ.

Because the ILP returned true, there is an integer vector x satisfying our inequalities. If $x_i = 0$, assign x_i to be false. Otherwise, assign x_i to be true.

To show that this is a satisfying assignment, we need to show that it satisfies all clauses. Suppose for

contradiction the clause C_j involving variables $x_{i_1}, x_{i_2}, x_{i_3}$ is not satisfied, meaning every literal is false. By our construction, this is because the ILP told us that every positive literal $= 0$ and every negative literal $\neq 0$. Because we had constraints $x_i \geq 0$ and $x_i \leq 1$, this means every negative literal $= 1$. Then,

$$\sum_{k=1,2,3} \begin{cases} x_{i_k} & \text{if the literal } x_{i_k} \text{ appears in } C_j \\ 1 - x_{i_k} & \text{if the literal } \neg x_{i_k} \text{ appears in } C_j \end{cases} = \sum_{k=1,2,3} \begin{cases} 0 & \text{if the literal } x_{i_k} \text{ appears in } C_j \\ 0 & \text{if the literal } \neg x_{i_k} \text{ appears in } C_j \end{cases} = 0,$$

which is a contradiction.

4. Reduce to decision

NP is a set of decision (yes/no) problems, but in practice we're often interested in optimization problems (instead of "is there a vertex cover of size k?" we usually want to "find the smallest vertex cover"). **Usually**, this isn't a problem, though; we'll see an example in this problem.

Let $\mathsf{VC_D}$ be the problem: Given a graph G and an integer k, return `true` if and only if G has a vertex cover of size k. Let $\mathsf{VC_O}$ be the problem: Given a graph G, return a list containing the vertices in a minimum size vertex cover.

(a) Show that $\mathsf{VC_D} \leq_P \mathsf{VC_O}$ (this is the easy direction). **Solution:**

On input G, k (for $k \leq n$) for $\mathsf{VC_D}$, run the library for $\mathsf{VC_O}$ on input G. Count the number of vertices in the output. If it is k or less, return `true`, otherwise return `false`.

If there is a vertex cover of size at most k, then there is a vertex cover of size k (just add vertices until you hit k). If there is not a vertex cover of size at most k, then a minimum one is larger, and so the $\mathsf{VC_D}$ algorithm will give a longer list, and the reduction will return false, as required.

(b) We'll now start working on the other reduction. Imagine someone came to you and said "See this vertex u, I promise it is in a minimum vertex cover." Use this promise to solve $\mathsf{VC_O}$ on a graph of size $n - 1$ instead of n. **Solution:**

If u is in a minimum vertex cover, then delete u and all edges incident to u from the graph G. Call the resulting graph $G - u$. Call the $\mathsf{VC_O}$ library on $G - u$. Return u along with the result of the library call.

Let S be a vertex cover of $G-u$. Observe that adding u gives a vertex cover of G, as every edge not incident to u was covered in $G - u$, and u was added to the vertex cover to cover all remaining edges. Moreover, we find a minimum vertex cover; We know that u is in a minimum vertex cover and removing u from any vertex cover for G gives a cover of $G - u$; a smaller cover of G including u would give us a smaller cover for $G - u$, but we called the $\mathsf{VC_O}$ library which gives us the minimum.

(c) Now imagine the same person said "See this vertex v, I promise it is **not** in any minimum vertex cover." Use this promise to solve $\mathsf{VC_O}$ on a graph of size at most $n - 1$ instead of n. **Solution:**

If v is not in the vertex cover, then all neighbors w of v must be in the cover (otherwise, we would not cover the edge (v, w)). So we delete v and all its neighbors denoted $N(v)$ from the graph. Then we can run VC_0 on the graph $G - v - N(v)$ and we return the result along with all vertices in $N(v)$.

Let S be a minimum vertex cover of G. Since S does not contain v by our assumption, then every neighbor $w \in N(v)$ has an edge (v, w) that needs to be covered, which means every neighbor must be in S. Then all edges coming out of v and its neighbors are covered, so we only need to solve the minimum vertex cover on the graph minus these vertices.

(d) Use the ideas from the last two parts to show $\mathsf{VC_O} \leq_P \mathsf{VC_D}$. **Solution:**

```
1: function MinVertexCover(G)
2:     Call VC_D library for all values of k until you find the size of the min vertex cover of G.
3:     Pick an arbitrary vertex u.
4:     if VC_D library says YES on G − u, k − 1 then
5:         return {u} ∪ MinVertexCover(G − u)
6:     elsereturn N(u) ∪ MinVertexCover(G − u − N(u))        ▷ N(u) is the neighbors of u
```

We will skip the proof of correctness, as it is mostly combining the prior parts.

For efficiency, observe that we need polynomial work and $n + 1$ library calls in each recursive call, and each recursive call reduces the problem size by at least 1, so we need at most n recursive calls. Thus the reduction is polynomial.

5. Another Reduction

Consider an undirected graph G, where each vertex has a non-negative integer number of pebbles. A single *pebbling move* consists of removing two pebbles from a vertex and adding one pebble to an adjacent vertex, where we can choose which adjacent vertex. A pebbling move can only be done on a vertex that already has at least two pebbles, and it will always decrease the total number of pebbles in the graph by exactly one. Our goal is to remove as many pebbles as we can. Observe that at best, we'll have at least one pebble remaining in the graph.

Define the PEBBLE problem as the following problem:

`Input`: An undirected graph and the number of pebbles at each vertex `Output`: `true` if there is a sequence of pebbling moves that leaves exactly one pebble in the graph, `false` otherwise.

Define the Hamiltonian Path Problem as the following problem:

Input: An undirected graph.

Output: `true` if there exists a path in the graph visiting every vertex exactly once, `false` otherwise.

Given that the Hamiltonian Path Problem is NP-complete, show that PEBBLE is as well. You may assume that the total number of pebbles in a graph is polynomial in terms of the size of the graph.

Hint: A single pebbling move can be represented as an ordered pair of vertices (u, v) where we take two pebbles from u and place one pebble in its neighbor v. A sequence of pebbling moves can be represented by a sequence of these pairs. Is there any way we can order these pairs nicely?

5.1. Read and Understand the Problem

Read the problem and answer these quick-check-questions.

Make sure you understand PEBBLE and the Hamiltonian Path Problem.

- What is the input type?
- What is the output type?
- Are any words in the problem technical terms? Do you know them all?

Solution:

PEBBLE

- Input: An undirected graph where each vertex is labeled with a non-negative integer
- Output: Boolean, can we take all but one pebble from the graph?

Hamiltonian Path Problem

- Input: An undirected graph

- Output: Boolean, is there a path that visits all vertices exactly once?

You're going to design a reduction – what will that reduction look like?

- Which problem are use solving, and which problem are you assuming you have an algorithm for? Make sure your reduction is "going the right direction"
- What is the output type for your reduction?

Solution:

We assume we have an algorithm for the PEBBLE problem. Given a Hamiltonian Path problem, try to turn it into a PEBBLE problem.

Output of Hamiltonian Path Problem returns `true` if and only if there is a path that visits all the vertices.

Output of PEBBLE problem is `true` if and only if there is a sequence of pebble moves that removes all but one pebble.

5.2. Design the Reduction

Now write a reduction. Remember a reduction is an algorithm! It often helps to think about the "certificates" (the thing that makes it a YES instance), and transform from one type of certificate to the other.

Solution:

Let G be the graph given in the Hamiltonian Path Problem. Let's say that G has n vertices.

Observe there are n possible starting vertices, so it's sufficient to check if there's a Hamiltonian Path for each of these n possible starting vertices.

Let that starting vertex be v_1.

Let $p(v)$ be the number of pebbles at vertex v. Define $p(v_1) = 2$ and $p(v) = 1$ otherwise (i.e. all vertices start with one pebble except the starting vertex, which starts with an additional pebble).

We assert that there is a Hamiltonian Path if and only if any of these n PEBBLE problems is true.

5.3. Write The Proof

(a) to be NP-Complete, PEBBLE needs to be in NP. Argue that it is (this argument is usually only 2-3 sentences).

Solution:

Given a sequence of pebbling moves, we just have to check if it's valid and if it's long enough to remove enough pebbles. At most this will take polynomial time in terms of the number of pebbles.

(b) Show your reduction is correct. Remember you need to prove two implications **and** that the running time is polynomial.

Solution:

First Implication:

Suppose there is a Hamiltonian Path $v_1, ..., v_n$.

Then consider the sequence of pebble moves: $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$. Observe for any v_i where $1 \leq i < n$, since v_i, v_{i+1} is in the Hamiltonian Path, then (v_i, v_{i+1}) must be an edge in G. Since (v_i, v_{i+1}) is the first pebble move that takes away pebbles from v_i, and since v_i starts with at least one pebble, then v_i

has all of its starting pebbles when we attempt to do that pebble move. In the case that $i = 1$, then v_i has enough pebbles to do the move since it starts with two pebbles. In the case that $1 < i < n$, then (v_{i-1}, v_i) was the previous pebble move, so v_i just gained a pebble and started with one pebble. So v_i has at least two pebbles and has enough to pebbles to make the pebble move.

After this sequence, observe v_n has not lost any pebbles, so it still has its starting pebble, and it also just gained a pebble from the move (v_{n-1}, v_n), so it has two pebbles. Since the graph started with $n+1$ pebbles and we made $n-1$ moves, there are only two pebbles left, and v_n has both of them. Then, simply add the move (v_n, v_{n-1}), which is valid since v_n has two pebbles and the edge (v_n, v_{n-1}) exists since (v_{n-1}, v_n) was a valid move. Now we have on pebble left.

So there is indeed a sequence of pebble moves removing all but one pebble.

Second Implication:

Suppose we have a sequence of pebble moves removing all but one pebble. We need to show that there also exists a Hamiltonian Path.

Since we start with $n + 1$ pebbles, and each move removes one pebble, this sequence must have exactly n moves. Denote the pebbling sequence as $(v_1, w_1), \ldots, (v_n, w_n)$ where v_i sends a pebble to w_i at step i.

Define the preposition $P(k)$ for $0 \leq k < n$ to be true iff after k moves, for $i = 1 \ldots, k$, v_i has zero pebbles, v_{k+1} 2 pebbles and $v_{i+1} = w_i$. Furthermore all vertices $v_1, \ldots, v_{k+1}$ are distinct. We prove by induction on k.

For $k = 0$, we trivially satisfy that v_1 is distinct.

Suppose $P(k-1)$ holds for some $0 < k - 1 < n - 1$, then we look at the kth move (v_k, w_k). We know that the $k-1$st move was (v_{k-1}, v_k) and v_k has two pebbles, so now consider all possible moves to w_k. We know that since we still have at least another move (v_{k+1}, w_{k+1}) left that w_k can only be v_{k+1}, or otherwise we only have at most 1 pebble at v_{k+1}. Furthermore $w_k = v_{k+1}$ must be a vertex distinct from v_i for $1 \leq i < k$ since these all had 0 pebbles up to move $k - 1$ and can only get 1 more pebble from the kth move. Finally we can see that v_k will now have 0 pebbles after the kth move.

Then by induction, the first $n - 1$ moves can be written as the sequence $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$.

Since this removes $n-1$ pebbles, there are 2 pebbles remaining after this sequence and by induction again the last vertex v_n contains two pebbles. Also all the vertices are distinct. Hence this sequence of vertices $v_1, \ldots, v_n$ form a Hamiltonian path as (v_i, v_{i+1}) are all valid edges for each $i < n$.

Polynomial Time: From the reduction, we run the PEBBLE algorithm n times, once for each start vertex, so this only contributes a polynomial factor. Additionally, to modify the graph each time, we simply label each vertex in constant time, which takes linear time to do so. So overall, this runtime is polynomial.

6. Vertex Cover and Independent Set

Define IND-SET as follows:
Input: An undirected graph G and a positive integer k
Output: `true` if there is an independent set in G of size at least k, `false` otherwise.

Define VER-COVER as follows:
Input: An undirected graph G and a positive integer k
Output: `true` if there is an vertex cover in G of size at most k, `false` otherwise.

Prove that VER-COVER is NP-complete using IND-SET.

6.1. Read and Understand the Problem

Read the problem and answer these quick-check-questions.

Make sure you understand IND-SET and VER-COVER.

- What is the input type?
- What is the output type?
- Are any words in the problem technical terms? Do you know them all?

Solution:

For VER-COVER

- input: a graph
- output: true or false
- A vertex cover is a set $S \subset G$ of vertices for every edge (u, v): $u \in S$ or $v \in S$

You're going to design a reduction – what will that reduction look like?

- Which problem are you solving, and which problem are you assuming you have an algorithm for? Make sure your reduction is "going the right direction"
- What is the output type for your reduction?

Solution:

For the reduction

- We want to show that IND-SET $\leq$ VER-COVER. Assume we have an algorithm for VER-COVER . We're trying to solve IND-SET.
- output of the reduction: a boolean which is the answer to the IND-SET (which we get by calling VER-COVER like a library function)

6.2. Design the Reduction

Now write a reduction. Remember a reduction is an algorithm! It often helps to think about the "certificates" (the thing that makes it a YES instance), and transform from one type of certificate to the other.

Solution:

The idea is taking the complement.

For any graph $G = (V, E)$, S is an independent set if and only if $V - S$ is a vertex cover.

And using the format given in the problem definition, there is an vertex cover in G of size k if and only if there is an independent set in G of size $|V| - k$.

6.3. Write The Proof

(i) to be NP-Complete, IND-SET needs to be in NP. Argue that it is (this argument is usually only 2-3 sentences).

Solution:

A verifier would take in the subset of k vertices that are a vertex cover. Given this set of vertices, a verifier would check that these vertices are actually covering the graph (i.e. are endpoints to each edge in the graph). This will take time $\mathcal{O}(|E| + |V|)$, so it is polynomial time.

(ii) Show your reduction is correct. Remember you need to prove two implications **and** that the running time is polynomial.

Solution:

Running Time: Our algorithm just returns the output of `VER-COVER` but with parameter $|V| - k$, which is polynomial time.

Correctness
Let $G = (V, E)$ and positive integer k be given by `IND-SET`.
Suppose G has an independent set of size at least k, we show that our reduction returns true. Let S be the independent set of size at least k, then every vertex in S touches at most one endpoint of every edge in G. So $V - S$ touches at least one endpoint of every edge of G. Hence $V - S$ is a vertex cover of size at most $|V| - k$ and thus our algorithm output true.

For the other direction suppose that our algorithm returns true, meaning there is a vertex cover of size at least $|V| - k$ in G. Let S be the vertex cover of size at least $|V| - k$, then S touches at least one endpoint of every edge in G. So $V - S$ touches at most one endpoint of ever edge in G. Hence $V - S$ is an independent set of size at least k.