Section 8: Solutions

1. A Fun Reduction

Define 5-SAT 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.

Prove that 5-SAT is NP-complete.

1.1. Read and Understand the Problem

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

Make sure you understand 5-SAT.

- 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 5-SAT

- 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 5-SAT is NP-complete, so we need to reduce an NP-complete problem to 5-SAT in polynomial time. We can assume we have an algorithm for 5-SAT. A good NP-complete problem to use is 3-SAT, so let's try to solve it using 5-SAT. 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 3-SAT (which we get by calling 5-SAT like a library function)

1.2. 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 3-SAT 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.

1.3. Write The Proof

(a) to be NP-Complete, 5-SAT needs to be in NP. Argue that it is (this argument is usually only 2-3 sentences).

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.

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

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).

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

2. A Tricky Reduction

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 k (or more), `false` otherwise.

And 3-SAT 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 IND-SET is NP-complete using 3-SAT.

2.1. Read and Understand the Problem

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

Make sure you understand IND-SET and 3-SAT.

- 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 IND-SET

- input: a graph
- output: true or false
- An independent set is a set of vertices so that G has no edges between any pair in the set.

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:

For the reduction

- We want to show that 3-SAT $\leq$ IND-SET. Assume we have an algorithm for IND-SET. We're trying to solve 3-SAT.
- output of the reduction: a boolean which is the answer to the 3-SAT (which we get by calling IND-SET 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:

The key idea is to think about solving 3-SAT as picking a literal from each clause and finding a truth assignment to make all of them true (where none of the literals we pick are in conflict, like x_i and $\neg x_i$). We want the independent set that IND-SET finds to correspond to those literals that make each clause true.

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

Let G be the graph we construct to input into IND-SET. Add one vertex to G for each literal in a clause. For each clause, connect the 3 literals to form a triangle. IND-SET will pick at most one vertex from each clause, which will be set to true. To ensure that no conflicting literals are picked, connect all pairs of vertices that correspond to complementary literals. Let k be the number of clauses m.

Our IND-SET instance is: The graph G described above.

2.3. Write The Proof

(a) 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 independent. Given this set of vertices, a verifier would check that these vertices are actually independent (i.e. they are not adjacent). This will take time $\mathcal{O}(E+V)$, 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: Our algorithm adds every clause to the graph, adds edges for the 3 literals in each clause, and adds edges connecting complementary literals, so the running time to create the instance is polynomial (and we call the library only once, which is also at most polynomial).

Correctness
Let ϕ_3 be our 3-SAT instance and G be our IND-SET 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, it gives us that at least one literal in every constraint is true, and no conflicting literals are both true. To get the assignment that satisfies G, for each constraint, choose one of the true literals, and add the corresponding vertex to the independent set. This gives an independent set of the required size.

Conversely, suppose that our reduction returns true, and therefore ϕ_3 was satisfiable. Suppose there is an independent set of at least the size of the number of constraints. Because of the "in-constraint" edges in G, an independent set has at most one per constraint. Thus every constraint has exactly one vertex in the independent set. The variables of the ϕ_3 match to the chosen literals. No set literals could be conflicting because of the edges in G connecting corresponding literals, and we satisfy every constraint because we chose a good setting of one piece with the independent set. Therefore we have a satisfying assignment of ϕ_3!

3. 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 the 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 the 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 the 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 the minimum vertex cover." Use this promise to solve $\mathsf{VC_O}$ on a graph of size $n-1$ 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)).

(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.

4. 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.

Let the PEBBLE problem be, given an undirected graph, and the number of pebbles at each vertex, is there a sequence of pebbling moves that leaves exactly one pebble in the graph?

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?

4.1. 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 guarantees the existence of a path that visits all the vertices.

Output of PEBBLE problem guarantees the existence of a sequence of pebble moves that removes all but one pebble.

4.2. 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.

4.3. 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.

Observe that after we've made zero pebble moves, the only vertex with at least two pebbles is v_1 by construction, which has exactly two.

Suppose for sake of induction that after we've made $k-1$ moves, where $0 \leq k-1 < n-1$, there is exactly one vertex v_k with at least two pebbles and that vertex has exactly two pebbles. Then since $k-1 < n-1$, we still have more moves in our sequence. Since only v_k has at least two pebbles, our kth move must be from v_k to some other vertex, call it v_{k+1}. In the case that v_{k+1} has no pebbles, then no vertices will have two pebbles after this move, so we can't make moves, but since $k < n$ we still have moves and that's a contradiction. By the inductive hypothesis, v_{k+1} has less than two pebbles, so v_{k+1} had one pebble and after the kth move it has two pebbles. So after k moves, there is exactly one vertex, namely v_{k+1}, with at least two pebbles and that vertex has exactly two pebbles.

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. Then one of the pebbles has to be at v_n since that was the last move in the subsequence. Since there is still one more move, one of the pebbles must have two pebbles, and since v_n already has one, it must have the other. So v_n is the only vertex with pebbles after the first $n-1$ moves.

Since each of the other $n-1$ vertices start with a pebble and end with none after the first $n-1$ moves, there must be some move that removes their pebble. Since there are exactly $n-1$ moves in the beginning, there is a bijection between those $n-1$ vertices and the first $n-1$ moves corresponding to which vertex a move removes pebbles from. So all the vertices $v_1, ..., v_n$ are distinct.

Then consider the path $v_1, ..., v_n$, which visits each of the n vertices exactly once. Observe this is indeed a valid path since for $1 \leq i < n-1$, since v_i, v_{i+1} was a pebble move, then v_i, v_{i+1} must be an edge. So this is a Hamiltonian Path.

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.