CS 341: ALGORITHMS (S18) — LECTURE 21
MORE NP-COMPLETENESS

ERIC BLAIS

We finished the last lecture by showing that from Cook–Levin's theorem that 3Sat is **NP**complete, we can show that Clique is also **NP**-complete. This result (along with the simple proofs that the following languates are in **NP**) implies that all of the decision problems

IndepSet,
VertexCover,
SetCover, and
SetPacking

are also **NP**-complete. Today, we continue our exploration of **NP**-completeness by adding more problems to this list.

1. HamPath is NP-complete

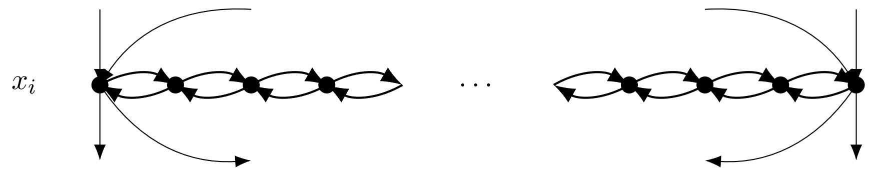
Let's show that HamPath is **NP**-complete. The easiest way to do so is by first showing that the analogous problem on directed graph is **NP**-complete. Define DirHamPath to be the problem where we are given a directed graph $G = (V, E)$ and must determine whether it contains a Hamiltonian path (= a path in G visiting each vertex of V exactly once).

Theorem 21.1. DirHamPath *is* **NP-complete**.

Proof idea. DirHamPath $\in$ **NP** since a verifier that demands the Hamiltonian path P through G as the certificate easily checks in time polynomial in the size of the graph whether P does visit each vertex in V exactly once and whether it is a valid path in G.

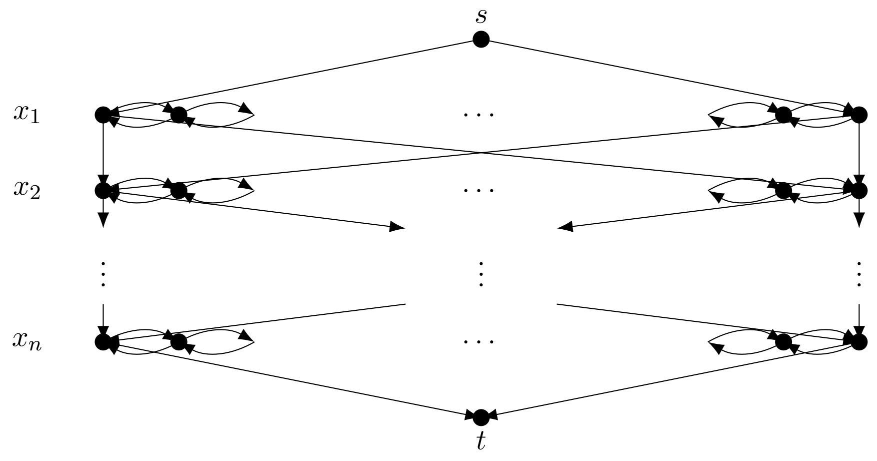
To show that DirHamPath is **NP**-complete, we now want to show that 3Sat $\leq_{\mathbf{P}}$ DirHamPath.

The main idea is that we will construct a *gadget* which is a graph that will correspond to the idea of assigning `True` or `False` values to each of the variables $x_1, \ldots, x_n$ in the original formula φ. The gadget is composed of a path of vertices for each variable x_i:

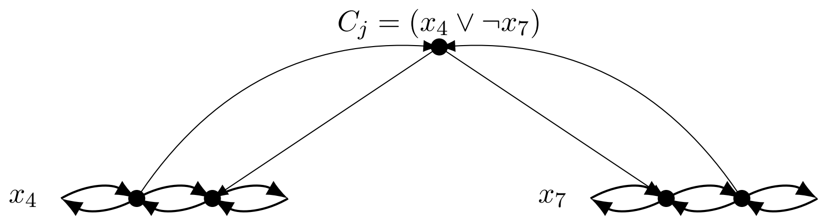


We have two choices in a Hamiltonian path: we either visit the vertices in that row left-to-right (which we will associate with setting of $x_i \leftarrow$ `True`) or right-to-left (which will correspond to $x_i \leftarrow$ `False`). The edges between each row mean that we can choose the assignment of `True` or `False` to each variable separately.

For each $i < n$, there are edges connecting the left- and right-most vertices of the path to x_i to the left- and right-most vertices of the path for x_{i+1}. We then complete the base gadget by adding a source vertex s with edges to the left- and right-most vertices of the path for x_1 and a target vertex t with edges going from the left- and right-most vertices of the path for x_n to t.



Now the gadget by itself has a Hamiltonian path from s to t. We want to add one more vertex for each clause in φ and connect it to the gadget in a way that we can visit the vertex corresponding to a clause only when our tour through the main gadget corresponds to an assignment to the variables that satisfies the clause. We can do this by constructing one vertex per clause and attaching it to the paths of the variables in the clause with the direction of the connection determined by whether that variable is negated or not in the clause, as in the following example:



Then the last subtle point is that we need to make sure that we attach those constructions with at least one spare vertex between any connections. When we complete the construction, we see that every satisfiable formula φ results in a graph G that has a Hamiltonian path: simply traverse x_i in the direction corresponding to the value assigned to it, and visit each clause vertex during the traversal of the row corresponding to one of its satisfying literals.

In the other direction, we want to argue that if G has a Hamiltonian path, the original formula φ is satisfiable. The key idea to doing this is to show that the only way that the vertex corresponding to a clause can be visited is as a detour through the path through the row corresponding to one of its satisfied literals: that's because path must start at s and end at t; if (x, c) edge is followed and (c, y) edge is not, then any path that goes through b must come from s_2 and reaches a dead-end. Therefore, we can read the direction through each row to find a satisfying assignment for φ. □

We are now ready to show that HamPath (on undirected graphs) is **NP**-complete.

Theorem 21.2. *HamPath is* **NP***-complete.*

Proof. HamPath $\in$ **NP** by essentially the same argument that we used for DirHamPath.

Let us now show that DirHamPath $\leq_{\mathbf{P}}$ HamPath. Given the directed graph $G = (V, E)$, let us build a graph $G' = (V', E')$ where for each vertex $v \in V$, we now create three vertices in V': v itself, v_{in}, and v_{out}. We will build E' by adding edges (v_{in}, v) and (v, v_{out}) for each $v \in V$, and for each $(u, v) \in E$ by adding the edge (u_{out}, v_{in}) in E'.

With this construction, if there is a Hamiltonian path $v^{(0)}, v^{(1)}, \ldots, v^{(n)}$ in G, then the path $v_{in}^{(0)}, v^{(0)}, v_{out}^{(0)}, v_{in}^{(1)}, \ldots, v_{out}^{(n)}$ is a Hamiltonian path in G'.

And if we have a Hamiltonian path in G', then we must have a Hamiltonian path in G as well... though this is not completely obvious! (Exercise: why is this claim true?) $\square$

2. SubsetSum is NP-complete

We can also use a reduction from 3Sat to show that SubsetSum is **NP**-hard. The key insight for this reduction is that we will want to use *really* big numbers in the reduction.

Theorem 21.3. SubsetSum *in* **NP**-*complete.*

Proof. We already saw in the last lecture that SubsetSum $\in$ **NP**. We complete the proof by showing that 3Sat $\leq_{\mathbf{P}}$ SubsetSum.

Given an instance φ of the 3Sat problem on n variables and with m clauses, we want to construct a set of numbers that we will turn into an instance of the SubsetSum problem. Let's start by building $2n$ numbers: one for each possible literal $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$. Each number will have m digits: digit j of x_i will be 1 if x_i is part of the jth clause and 0 otherwise, and likewise for all the negations. So we end up with numbers that look like this:

	C_1	C_2	C_3	$\cdots$	C_m
$x_1 =$	0	0	1	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1
$\neg x_2 =$	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0
$T =$	≥ 1	≥ 1	≥ 1	$\cdots$	≥ 1

If we have a satisfying assignment to φ, then we can choose a subset of those numbers such that each digit is ≥ 1—specifically, either 1, 2, or 3. We can add another $2c$ numbers to act as "slack" numbers that we can add to other numbers to make each digit *exactly* 4 when each constraint is satisfied. Our set of numbers and our target number now look like this:

	C_1	C_2	C_3	$\cdots$	C_m
$x_1 =$	0	0	1	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1
$\neg x_2 =$	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0
$s_{1,1} =$	1	0	0	$\cdots$	0
$s_{1,2} =$	2	0	0	$\cdots$	0
$s_{2,1} =$	0	1	0	$\cdots$	0
$s_{2,2} =$	0	2	0	$\cdots$	0
$s_{3,1} =$	0	0	1	$\cdots$	0
$s_{3,2} =$	0	0	2	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$s_{m,1} =$	0	0	0	$\cdots$	1
$s_{m,2} =$	0	0	0	$\cdots$	2
$T =$	4	4	4	$\cdots$	4

There's one remaining issue that we need to handle: at the moment, nothing prevents us to choose the numbers corresponding to both x_i and $\neg x_i$ in our set of numbers that we choose to try to reach the target number. How can we disallow this? We can do it by adding extra digits to the numbers and to the target where the target can be reached only when we choose exactly one of x_i or $\neg x_i$ for each $i = 1, 2, \ldots, n$. The final construction looks like this:

	C_1	C_2	C_3	$\cdots$	C_m	x_1	x_2	$\cdots$	x_n
$x_1 =$	0	0	1	$\cdots$	0	1	0	$\cdots$	0
$\neg x_1 =$	1	0	0	$\cdots$	0	1	0	$\cdots$	0
$x_2 =$	0	0	0	$\cdots$	1	0	1	$\cdots$	0
$\neg x_2 =$	1	1	0	$\cdots$	0	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\neg x_n =$	0	0	1	$\cdots$	0	0	0	$\cdots$	1
$s_{1,1} =$	1	0	0	$\cdots$	0	0	0	$\cdots$	0
$s_{1,2} =$	2	0	0	$\cdots$	0	0	0	$\cdots$	0
$s_{2,1} =$	0	1	0	$\cdots$	0	0	0	$\cdots$	0
$s_{2,2} =$	0	2	0	$\cdots$	0	0	0	$\cdots$	0
$s_{3,1} =$	0	0	1	$\cdots$	0	0	0	$\cdots$	0
$s_{3,2} =$	0	0	2	$\cdots$	0	0	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$s_{m,1} =$	0	0	0	$\cdots$	1	0	0	$\cdots$	0
$s_{m,2} =$	0	0	0	$\cdots$	2	0	0	$\cdots$	0
$T =$	4	4	4	$\cdots$	4	1	1	$\cdots$	1

This construction guarantees that there is a subset of the integers that sums up to T if and only if there is a satisfying assignment to φ. □

3. **NP**-complete problems are everywhere

There are a ton of **NP**-complete problems that have appeared in all areas of computer science. Some examples include the following.

Graph Colourability.: Vizing's theorem states that for every graph G with maximum degree Δ, either Δ or $\Delta + 1$ colours are necessary and sufficient to colour the edges of the graph so that no two edges that share a common vertex endpoint have the same colour. Given a graph G, can we determine if Δ or $\Delta + 1$ colours are required for that graph? That's an **NP**-complete problem!

Super Mario.: Many video games, including Super Mario and Tetris, involve decision problems (e.g., can you go from s to t in the map) that are **NP**-complete.

0-1 Linear programming.: A really powerful algorithmic tool that we do not cover in this class is *linear programming.* The usefulness of this technique stems from the fact that there are many natural problems that can be stated in the form

$$\begin{aligned} &\text{maximize} \sum_i x_i \text{subject to} && x_i + x_j \le a_{i,j} \forall i, j \\ &0 \le x_i \le 1 && \forall i \end{aligned}$$

and that such problems can be solved in polynomial time. But if we consider the very slight modification of linear programs where we require the variables to take only the values 0 or 1 (instead of any real value between them), then the (associated decision) problem becomes **NP**-complete.

Number theory.: Here's a simple basic number theoretic question: given some positive integers a and b, does there exist x such that $x^2 \equiv a \pmod{b}$? This problem is **NP**-complete.

We could go on and on—with problems in databases, computational geometry, networking, scheduling jobs on servers, comparing regular expressions, etc. In fact, there is a whole book devoted to collecting many of the fundamental **NP**-complete problems: Garey and Johnson's *Computers and Intractability: A Guide to the theory of* **NP***-completeness.* If you ever want to see if a problem X that you are considering is **NP**-complete, that's the right place to look first for a related problem A that is **NP**-complete and that would let you obtain an easy proof that $A \le_{\mathbf{P}} X$ to show that X is **NP**-complete as well.

4. Closing thoughts

There is lots more we could say about the class **NP**. Here are perhaps the four most important questions in the FAQ about **NP** that we have not yet covered.

4.1. **Do NP-complete and NP-hard mean the same thing?** You will often see a problem labelled as "**NP**-hard" instead of **NP**-complete. This does *not* mean the same thing: it's a slightly weaker statement:

Definition 21.4. The decision problem X is **NP***-hard* if every problem $A \in \mathbf{NP}$ satisfies $A \le_{\mathbf{P}} X$.

(In other words, for **NP**-hardness we do not need to show that X is in **NP**; for **NP**-completeness we *do* need to show that.) There are some problems that are **NP**-hard but not **NP**-complete, but almost all of the "hard" problems that you will encounter (and all of the ones we have covered so far in the course) are in **NP**, so for this class we will *always* aim to show that a problem is **NP**-complete, not just **NP**-hard.

4.2. **Can we use other types of reductions to show that a problem is hard?** Yes! There's a more general type of polynomial-time reduction between decision problems known as *Cook reductions*.

Definition 21.5. There is a *Cook reduction* from A to B if we can design a polynomial-time algorithm F that, using a polynomial-time algorithm for B as a black-box, solves A in polynomial time.

This means that if we can show a Cook reduction from some problem Hard that we know has no polynomial-time algorithm to the problem X, then it means that there is no polynomial-time algorithm for X either.

Note that if $A \leq_{\mathbf{P}} B$ then there is a Cook reduction from A to B, but the converse is not known to be true. Let's discuss this a bit more in the next question.

4.3. **Are there problems that we can't verify efficiently?** Yes! Or, at least, there are problems that we *don't know* how to verify in polynomial time at the moment. Consider for example the problem we obtain by negating the Clique problem.

Definition 21.6. coClique: Given a graph $G = (V, E)$ and a positive integer k, is the largest clique of G of size $< k$?

Here it is easy to give a certificate that would help convince a verifier that the answer is `No`—but we have to provide certificates when the answer is `Yes` instead! Is there any way to define a certificate that would help convince a verifier that a graph does *not* contain any clique of size k? It's certainly not enough to give a particular subgraph of size k and show that this subgraph is not a clique—the clique could be elsewhere in the graph! In fact, we currently don't know of *any* way to efficiently verify the coClique problem.

The class of problems that have polynomial-time verifiers for `No` inputs instead of `Yes` inputs is known as **coNP**. It is believed that $\mathbf{NP} \neq \mathbf{coNP}$... except that once again we can't hope to prove this statement before we first settle the famous **P** vs. **NP** problem.

4.4. **Are there problems that are even harder than the ones in NP?.** Yes, lots! There are even *undecidable* problems—problems that cannot be solved by *any* algorithm, no matter how efficient or inefficient. We'll talk about this in a bit more detail in one of the last two lectures of the class.