15-451/651: Design & Analysis of Algorithms March 21, 2019
Lecture #16: NP Completeness last changed: March 14, 2019

In the past few lectures we have looked at increasingly more expressive problems solvable using efficient algorithms. In this lecture we introduce a class of problems that are so expressive — they are able to model *any* problem in an extremely large class called **NP**— that we believe them to be *intrinsically unsolvable by polynomial-time algorithms.* These are the **NP-complete** problems. What is particularly surprising about this class is that they include many problems that at first glance appear to be quite benign.

Specific topics in this lecture include:

- Reductions and expressiveness
- Formal definitions: decision problems, **P** and **NP**.
- Circuit-SAT and 3-SAT
- Examples of showing **NP**-completeness.

1 Reductions and Expressiveness

In the last few lectures we have seen a series of increasingly more expressive problems: network flow, min cost max flow, and finally linear programming. These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up with important tools we can use to solve other problems.

To talk about this a little more precisely, it is helpful to make the following definitions:

Definition 1 *We say that an algorithm runs in* **Polynomial Time** *if, for some constant c, its running time is $O(n^c)$, where n is the size of the input.*

In the above definition, "size of input" means "number of bits it takes to write the input down". So, to be precise, when defining a problem and asking whether or not a certain algorithm runs in polynomial time, it is important to say how the input is given.

> **Example:** Think about why the basic Ford-Fulkerson algorithm is *not* a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms *are* polynomial-time?

Definition 2 *A problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B.*[1] *Problem A is* **poly-time equivalent** *to problem B ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.*

For instance, we gave an efficient algorithm for Bipartite Matching by showing it was poly-time reducible to Max Flow. Notice that it could be that $A \leq_p B$ and yet our fastest algorithm for solving problem A might be slower than our fastest algorithm for solving problem B (because our reduction might involve several calls to the algorithm for problem B, or might involve blowing up the input size by a polynomial but still nontrivial amount).

[1]You can loosely think of $A \leq_p B$ as saying "A is no harder than B, up to polynomial factors."

1.1 Decision Problems and Karp Reductions

We consider *decision problems*: problems whose answer is YES or NO. E.g., "Does the given network have a flow of value at least k?" or "Does the given graph have a 3-coloring?" For such problems, we split all instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We put any ill-formed instances into the NO category.

In this lecture, we seek reductions (called *Karp reductions*) that are of a special form:

Many-one reduction (a.k.a. Karp reduction) from problem A to problem B: To reduce problem A to problem B we want a function f that maps arbitrary instances of A to instances of B such that:

1. if x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. if x is a NO-instance of A then $f(x)$ is a NO-instance of B.
3. f can be computed in polynomial time.

So, if we had an algorithm for B, and a function f with the above properties, we could using it to solve A on any instance x by running it on $f(x)$. [2]

2 Definitions: P, NP, and NP-Completeness

We can now define the complexity classes **P** and **NP**. These are both classes of decision problems.

Definition 3 **P** *is the set of decision problems solvable in polynomial time.*

E.g., the decision version of the network flow problem: "Given a network G and a flow value k, does there exist a flow $\geq k$?" belongs to **P**.

But there are other problems we don't know how to efficiently solve. Some of these problems may be really uncomputable (like the HALTING PROBLEM that you probably saw in 15-251). Others, like the TRAVELING SALESMAN PROBLEM, have algorithms that run in $2^{O(n)}$ time. Yet others, like FACTORING, have algorithms that run in $2^{O(n^{1/3})}$ time. How to refine the landscape of these problems, to make sense of it all?

Here's one way. Many of the problems we would like to solve have the property that if someone handed us a solution, we could at least check if the solution was correct. For instance the TRAVELING SALESMAN PROBLEM asks: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" We may not know how to find such a tour quickly, but if someone gave such a tour to us, we could easily check if it satisfied the desired conditions (visited all the vertices and had total length at most k). Similarly, for the 3-COLORING problem: "Given a graph G, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?" we don't know of any polynomial-time algorithms for solving the problem but we could easily check a proposed solution if someone gave one to us. The class of problems of this type — namely, if the answer is YES, then there exists a polynomial-length proof that can be checked in polynomial time — is called **NP**.

[2] Why Karp reductions? Why not reductions that, e.g., map YES-instances of A to NO-instances of B? Or solve two instances of B and use that answer to solve an instance of A? Two reasons. Firstly, Karp reductions give a stronger result. Secondly, using general reductions (called Turing reductions) no longer allows us to differentiate between **NP** and co-**NP**, say; see Section 8 for a discussion.

Definition 4 **NP** *is the set of decision problems that have polynomial-time* verifiers. *Specifically, problem Q is in* **NP** *if there is a polynomial-time algorithm $V(I, X)$ such that:*

- *If I is a YES-instance, then there exists X such that $V(I, X) =$ YES.*
- *If I is a NO-instance, then for all X, $V(I, X) =$ NO.*

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance, not the combined size of the instance and solution).

The second input X to the verifier V is often called a *witness*. E.g., for 3-coloring, the witness that an answer is YES is the coloring. For factoring, the witness that N has a factor between 2 and k is a factor. For the Traveling Salesman Problem: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" the witness is the tour. All these problems belong to **NP**. Of course, any problem in **P** is also in **NP**, since V could just ignore X and directly solve I. So, $\mathbf{P} \subseteq \mathbf{NP}$.

A huge open question in complexity theory is whether $\mathbf{P} = \mathbf{NP}$. It would be quite strange if they were equal since that would mean that any problem for which a solution can be easily *verified* also has the property that a solution can be easily *found*. So most people believe $\mathbf{P} \neq \mathbf{NP}$. But, it's very hard to prove that a fast algorithm for something does *not* exist. So, it's still an open problem.

Loosely speaking, **NP**-complete problems are the "hardest" problems in **NP**, if you can solve them in polynomial time then you can solve any other problem in **NP** in polynomial time. Formally,

Definition 5 (NP-complete) *Problem Q is* **NP***-complete if:*

1. *Q is in* **NP***, and*
2. *For any other problem Q' in* **NP***, $Q' \leq_p Q$.*

So if Q is **NP**-complete and you could solve Q in polynomial time, you could solve *any* problem in **NP** in polynomial time. If Q just satisfies part (2) of Definition 5, then it's called **NP**-hard.

3 Circuit-SAT and 3-SAT (Read on your own)

The definition of **NP**-completeness would be useless if there were no **NP**-complete problems. Thankfully that is not the case. Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem in **NP**. Moreover, we must define the problem so that it is in **NP** itself. Here is a natural candidate: CIRCUIT-SAT.

Definition 6 Circuit-SAT: *Given a circuit of NAND gates with a single output and no loops (some of the inputs may be hardwired). Question: is there a setting of the inputs that causes the circuit to output 1?*

Theorem 7 *CIRCUIT-SAT is* **NP***-complete.*

Proof Sketch: First of all, CIRCUIT-SAT is clearly in **NP**, since you can just guess the input and try it. To show it is **NP**-complete, we need to reduce any problem in NP to CIRCUIT-SAT. By definition, this problem has a verifier program V that given an instance I and a witness W correctly outputs YES/NO in $b := p(|I|)$ time for a fixed polynomial $p()$. We can assume it only uses b bits of memory too. We now use the fact that one can construct a RAM with b bits of memory (including its stored program) and a standard instruction set using only $O(b \log b)$ NAND gates and a clock. By unrolling this design for b levels, we can remove loops and create a circuit that simulates what V computes within b time steps. We then hardwire the inputs corresponding to I and feed this into our CIRCUIT-SAT solver. ■

We now have one **NP**-complete problem. And it looks complicated. However, now we will show that a much simpler-looking problem, 3-SAT has the property that CIRCUIT-SAT $\leq_p$ 3-SAT.

Definition 8 3-SAT: *Given: a CNF formula (AND of ORs) over n variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. E.g., $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \ldots$. Goal: find an assignment to the variables that satisfies the formula if one exists.*

Theorem 9 *CIRCUIT-SAT $\leq_p$ 3-SAT. Hence 3-SAT is **NP**-complete.*

Proof: First of all, 3-SAT is clearly in **NP**, again you can guess the input and try it. Now, we give a Karp reduction from Circuit-SAT to it: i.e., a function f from instances C of CIRCUIT-SAT to instances of 3-SAT such that the formula $f(C)$ produced is satisfiable iff the circuit C had an input x such that $C(x) = 1$. Moreover, $f(C)$ should be computable in polynomial time, which among other things means we cannot blow up the size of C by more than a polynomial factor.

First of all, let's assume our input is given as a list of gates, where for each gate g_i we are told what its inputs are connected to. For example, such a list might look like: $g_1 = \mathsf{NAND}(x_1, x_3)$; $g_2 = \mathsf{NAND}(g_1, x_4)$; $g_3 = \mathsf{NAND}(x_1, 1)$; $g_4 = \mathsf{NAND}(g_1, g_2)$; .... In addition we are told which gate g_m is the output of the circuit.

We will now compile this into an instance of 3-SAT as follows. We will make one variable for each input x_i of the circuit, and one for every gate g_i. We now write each NAND as a conjunction of 4 clauses. In particular, we just replace each statement of the form "$y_3 = \mathsf{NAND}(y_1, y_2)$" with:

$$\begin{array}{lll}
 & (y_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\
\text{AND} & (y_1 \text{ OR } \overline{y}_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 0 \text{ and } y_2 = 1 \text{ then we must have } y_3 = 1 \\
\text{AND} & (\overline{y}_1 \text{ OR } y_2 \text{ OR } y_3) & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 0 \text{ then we must have } y_3 = 1 \\
\text{AND} & (\overline{y}_1 \text{ OR } \overline{y}_2 \text{ OR } \overline{y}_3). & \leftarrow \text{if } y_1 = 1 \text{ and } y_2 = 1 \text{ we must have } y_3 = 0
\end{array}$$

Finally, we add the clause (g_m), requiring the circuit to output 1. In other words, we are asking: is there an input to the circuit *and* a setting of all the gates such that the output of the circuit is equal to 1, *and* each gate is doing what it's supposed to? So, the 3-CNF formula produced is satisfiable if and only if the circuit has a setting of inputs that causes it to output 1. The size of the formula is linear in the size of the circuit. Moreover, the construction can be done in polynomial (actually, linear) time. So, if we had a polynomial-time algorithm to solve 3-SAT, then we could solve circuit-SAT in polynomial time too. ■

4 Search versus Decision

Technically, a polynomial-time algorithm for CIRCUIT-SAT or 3-SAT just tells us if a solution exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question of CIRCUIT-SAT to actually find a satisfying assignment? If we can do this, then we can use it to actually *find* the coloring or *find* the tour, not just smugly tell us that there is one. The problem of actually finding a solution is often called the *search* version of the problem, as opposed to the *decision* version that just asks whether or not the solution exists. That is, we are asking: can we reduce the search version of the CIRCUIT-SAT to the decision version?

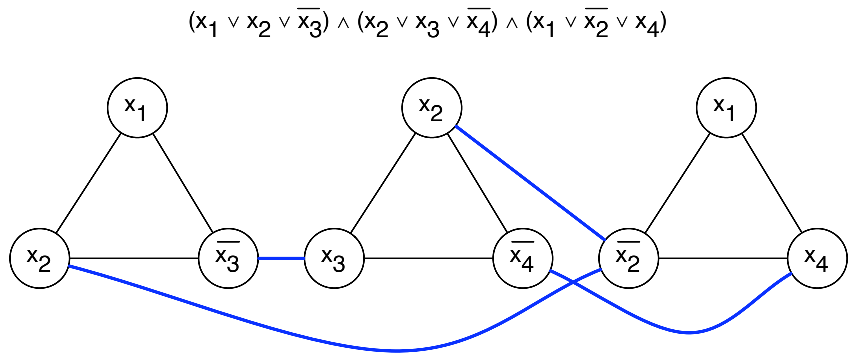
It turns out that in fact we can, by essentially performing binary search. In particular, once we know that a solution x exists, we want to ask: "how about a solution whose first bit is 0?" If, say, the answer to that is YES, then we will ask: "how about a solution whose first two bits are 00?" If, say, the answer to that is NO (so there must exist a solution whose first two bits are 01) we will then ask: "how about a solution whose first three bits are 010?" And so on. The key point is that we can do this using a black-box algorithm for the decision version of CIRCUIT-SAT as follows: we can just set the first few inputs of the circuit to whatever we want, and feed the resulting circuit to the algorithm. This way, using at most n calls to the decision algorithm, we can solve the search problem too.

5 Warm up: Independent Set, Vertex Cover, and Set Cover

An Independent Set in a graph is a set of nodes no two of which have an edge. E.g., in a 7-cycle, the largest independent set has size 3, and in the graph coloring problem, the set of nodes colored red is an independent set. The Independent Set problem is: given a graph G and an integer k, does G have an independent set of size $\geq k$?

Theorem 10 Independent Set *is* **NP**-*complete.*

Proof: We reduce from 3SAT. Let ϕ be an arbitrary 3SAT instance with clauses $c_1, \ldots, c_k$ and variables $x_1, \ldots, x_n$. We construct a graph G_ϕ that has a triangle T_i for each clause c_i, where the nodes of triangle T_i are labeled with the terms of c_i (a term is a variable or its negation). Between triangles, we connect up terms that are the negations of each other. For example:


$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$
x_1
x_2
$\overline{x_3}$
x_2
x_3
$\overline{x_4}$
x_1
$\overline{x_2}$
x_4

If ϕ is satisfiable, there is a true term in each triangle, and these true terms are all consistent with one another (none is the negation of another). This means that we can pick that node from each triangle, leading to an independent set of size k. If there is an independent set of size k, then we can set each of those terms to true in a consistent way, leading to a satisfying assignment. ■

A *vertex cover* in a graph is a set of nodes such that every edge is incident to at least one of them. For instance, if the graph represents rooms and corridors in a museum, then a vertex cover is a set of rooms we can put security guards in such that every corridor is observed by at least one guard. In this case we want the smallest cover possible. The Vertex Cover problem is: given a graph G and an integer k, does G have a vertex cover of size $\leq k$?

Theorem 11 VERTEX COVER *is* **NP**-*complete.*

Proof: If C is a vertex cover in a graph G with vertex set V, then $V - C$ is an independent set. Also if S is an independent set, then $V - S$ is a vertex cover. So, the reduction from INDEPENDENT SET to VERTEX COVER is very simple: given an instance (G, k) for INDEPENDENT SET, produce the instance $(G, n - k)$ for VERTEX COVER, where $n = |V|$. In other words, to solve the question "is there an independent set of size at least k" just solve the question "is there a vertex cover of size $\leq n - k$?" So, VERTEX COVER is **NP**-Complete too. ■

Set cover is a generalization of vertex cover that comes up all the time. Let U be a set of items and $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a collection of subsets of U. The *Set Cover* problem asks whether we can choose a subset of $\leq k$ sets such that every $u \in U$ is in at least one of the chosen sets.

Theorem 12 SET COVER *is NP-complete.*

Proof: Set Cover is in NP since we can check whether a given choice of sets covers all the elements of V in polynomial time. We reduce from Vertex Cover. Let $G = (V, E), k$ be an arbitrary instance of VC. Construct a Set Cover instance where the $U = E$ and we have a set S_v for every $v \in V$ where S_v contains the edges that are incident to v. The idea is that picking S_v covers the incident edges, and this corresponds to picking v and covering the incident edges.

If there is a vertex cover of size $\leq k$, then every item in U is covered by choosing the sets corresponding to the vertices chosen in the vertex cover.If there is a set cover of size $\leq k$, then choosing the corresponding vertices in G gives a vertex cover of the same size. ■

6 Summary: Proving NP-completeness in 2 Easy Steps

If you want to prove that problem Q is NP-complete, you need to do two things:

1. Show that Q is in NP.

2. Choose some NP-hard problem P to reduce from. This problem could be 3-SAT or CLIQUE or INDEPENDENT SET or VERTEX COVER or any of the zillions of NP-hard problems known.

   Now you want to reduce **from** P **to** Q. In other words, given any instance I of P, show how to transform it into an instance $f(I)$ of Q, such that

   $$I \text{ is a YES-instance of } P \iff f(I) \text{ is a YES-instance of } Q.$$

   Note the "$\iff$" in the middle—*you need to show both directions.*

   You also need to show that the mapping $f(\cdot)$ can be done in polynomial time (and hence $f(I)$ has size polynomial in the size of the original instance I).

A common mistake is reducing from the problem Q to the hard problem P. Think about what this means. It means you can model your problem Q as a hard problem, and hence use an algorithm for a hard problem to solve your problem Q. However, that does not imply Q is hard. For example, you can model the problem of adding two numbers as a linear program or as 3-SAT, but that does not make addition complicated. You want to reduce *from* the hard problem P *to* your problem Q, this shows Q is "at least as hard" as P.

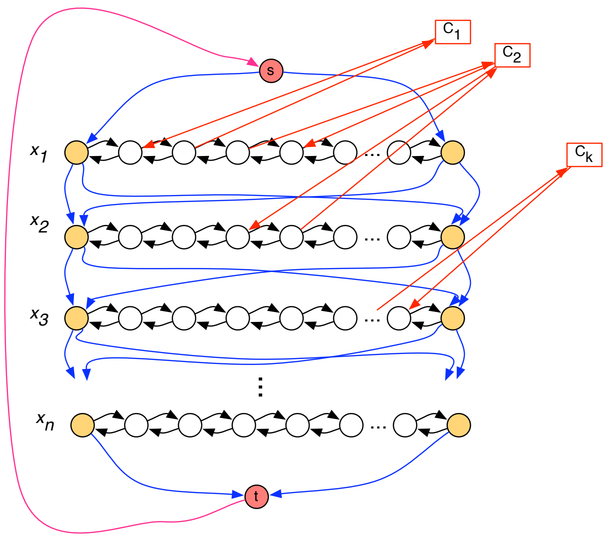
7 More Examples (Please Read on your Own)

7.1 Hamiltonian Cycle

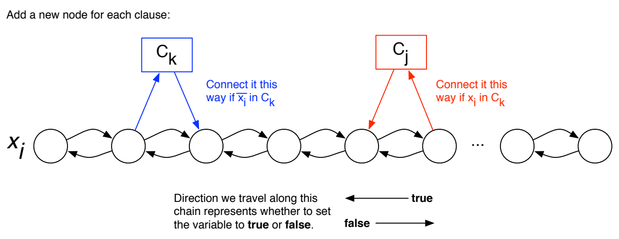
A *Hamiltonian cycle* is a cycle in a graph that visits every vertex exactly once. The Ham Cycle problem asks, given a directed graph G, is there a Hamiltonian cycle.

Theorem 13 Ham Cycle *is NP-complete*

Proof: Ham Cycle is in NP, since a certificate is just the cycle. We reduce from 3SAT. Let ϕ be an arbitrary 3SAT instance with clauses $c_1, \ldots, c_m$ and variables $x_1, \ldots, x_n$. Construct the following gadget that represents all possible truth assignments:



This includes a node (squares) for each clause. We connect the clauses up to the paths in an orientation that depends on whether the variable needs to be true or false to satisfy the clauses:


Add a new node for each clause:
Ck
Cj
Connect it this way if x̄i in Ck
Connect it this way if xi in Ck
xi
Direction we travel along this chain represents whether to set the variable to true or false.
true
false

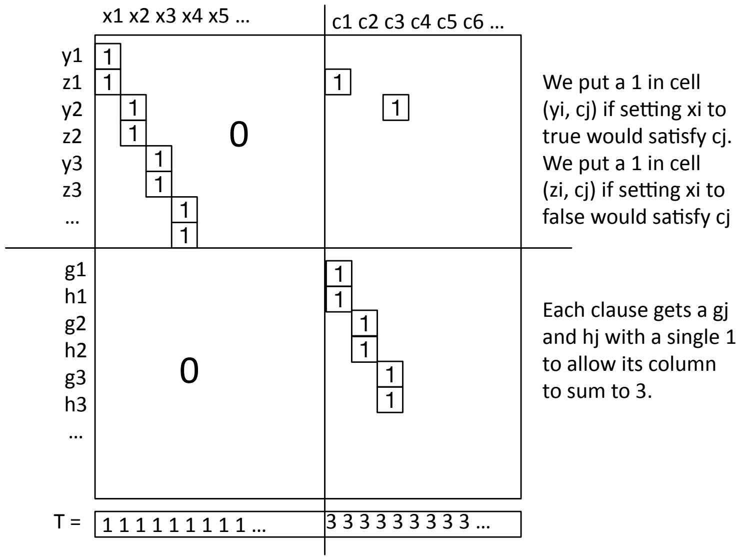
Each path through the rows represents a truth assignment, and we can take the detours to visit each clause node only iff we set the variables in such a way to satisfy them. ■

7.2 Subset Sum

The subset sum problem is this: Given integers $w_1, \ldots, w_n$ and T, does there exist a subset of the w_i integers that sums exactly to T? We saw a DP algorithm for this that ran in $O(nT)$ time.

Theorem 14 Subset Sum *is NP-complete.*

Proof: A certificate is simply the subset of the integers that sums to T. This has size strictly less than the input. We reduce from 3SAT. Let ϕ be an arbitrary 3SAT instance with clauses $c_1, \ldots, c_m$ and variables $x_1, \ldots, x_n$. We construct integers y_i and z_i for each variable x_i and g_j, h_j for each clause c_j. The idea will be that choosing y_i will mean that variable x_i should be true and choosing z_i will mean that it should be false. We construct these integers as follows (where each row below gives the digits of one of the integers):



x1 x2 x3 x4 x5 …
c1 c2 c3 c4 c5 c6 …
y1
z1
y2
z2
y3
z3
…
g1
h1
g2
h2
g3
h3
…
1
0
We put a 1 in cell (yi, cj) if setting xi to true would satisfy cj. We put a 1 in cell (zi, cj) if setting xi to false would satisfy cj
Each clause gets a gj and hj with a single 1 to allow its column to sum to 3.
T =
1 1 1 1 1 1 1 1 1 …
3 3 3 3 3 3 3 3 3 …

Since we are operating base-10, there is no carry. The only way we can sum to T is if we have chosen exactly 1 of y_i or z_i for each variable x_i. The only way we can sum to 3 in a column in the 2nd part of the matrix is if we've chosen at least one 1 in the upper right quadrant (since we can grab at most 2 from the g and h numbers). ■

7.3 CLIQUE

We now use the **NP**-completeness of 3-SAT to show that another problem, a natural graph problem called CLIQUE is **NP**-complete.

Definition 15 CLIQUE*: Given a graph G, find the largest clique (set of nodes such that all pairs in the set are neighbors). Decision problem: "Given G and integer k, does G contain a clique of size* $\geq k$*?"*

Theorem 16 CLIQUE *is* **NP**-*Complete.*

Proof: Note that CLIQUE is clearly in **NP**; the witness is the set of vertices that are all connected to each other via edges. Next, we reduce 3-SAT to CLIQUE. Specifically, given a 3-CNF formula F of m clauses over n variables, we construct a graph as follows. First, for each clause c of F we create one node for every assignment to variables in c that satisfies c. E.g., say we have:

$$F = (x_1 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge \ldots$$

Then in this case we would create nodes like this:

$$\begin{array}{llll}
(x_1 = 0, x_2 = 0, x_4 = 0) & (x_3 = 0, x_4 = 0) & (x_2 = 0, x_3 = 0) & \ldots \\
(x_1 = 0, x_2 = 1, x_4 = 0) & (x_3 = 0, x_4 = 1) & (x_2 = 0, x_3 = 1) & \\
(x_1 = 0, x_2 = 1, x_4 = 1) & (x_3 = 1, x_4 = 1) & (x_2 = 1, x_3 = 0) & \\
(x_1 = 1, x_2 = 0, x_4 = 0) & & & \\
(x_1 = 1, x_2 = 0, x_4 = 1) & & & \\
(x_1 = 1, x_2 = 1, x_4 = 0) & & & \\
(x_1 = 1, x_2 = 1, x_4 = 1) & & &
\end{array}$$

We then put an edge between two nodes if the partial assignments are consistent. Notice that the maximum possible clique size is m because there are no edges between any two nodes that correspond to the same clause c. We claim that the maximum size is m if and only if the original formula has a satisfying assignment.

Suppose the 3-SAT problem *does* have a satisfying assignment, then in fact there *is* an m-clique (just pick some satisfying assignment and take the m nodes consistent with that assignment).

For the other direction, we can either show that if there *isn't* a satisfying assignment to F then the maximum clique in the graph has size $< m$, or argue the contrapositive and show that if there is a m-clique in the graph, then there is a satisfying assignment for the formula. Specifically, if the graph has an m-clique, then this clique must contain one node per clause c. So, just read off the assignment given in the nodes of the clique: this by construction will satisfy all the clauses. So, we have shown this graph has a clique of size m iff F was satisfiable.

Finally, to complete the proof, we note that our reduction is polynomial time since the graph produced has total size at most quadratic in the size of the formula F ($O(m)$ nodes, $O(m^2)$ edges). Therefore Clique is **NP**-complete. ■

8 Co-NP (optional)

Just like we defined NP as the class of problems for which there were short proofs for YES-instances, we can define a class of problems for which there are short proofs/witnesses for NO-instances. Specifically, $Q \in$ **co-NP** if there exists a verifier $V(I, X)$ such that:

- If I is a YES-instance, for all X, $V(I, X) =$ YES,
- If I is a NO-instance, then there exists X such that $V(I, X) =$ NO,

and furthermore the length of X and the running time of V are polynomial in $|I|$.

For example, the problem Circuit-Equivalence: "Given two circuits C_1, C_2, do they compute the same function?" is in **co-NP**, because if the answer is NO, then there is a short, easily verified proof (an input x such that $C_1(x) \neq C_2(x)$). Or the problem co-Clique: "Given a graph G and a number K, is every clique in the graph of size at least K?" If the answer is NO, there is a short witness (a clique in G with fewer than K vertices). co-3-SAT: "Given a 3-CNF formula, does it have no satisfying assignments?" If the answer is NO, there is a short witness (a satisfying assignment).

It is commonly believed that **NP** does not equal **co-NP**. Note that if **P** = **NP**, then **NP** = **co-NP**, but the other implication is not known to be true. Again, one can define **co-NP**-complete problems. This is there using Karp reductions (as opposed to Turing reductions) becomes important.