CIS 320: Algorithms November 29-December 1, 2021

Lecture 22-23

Lecturer: Aaron Roth *Scribe: Aaron Roth*

NP Completeness I and II

Throughout this class, we have been focused on problems that we can solve efficiently with algorithms. But not every problem has an efficient solution — in fact, in a formal sense (that you will have seen in CIS 262), almost none of them have algorithmic solutions at all, regardless of run-time! In the next two lectures we'll focus on a broad class of problems that *are* algorithmically solvable, but *probably* do not have polynomial time algorithms. But this is currently conjectural. We'll describe the theory of NP completeness that is highly suggestive of this conjecture — but proving unconditionally that these problems have no polynomial time solutions (the "P vs. NP" problem) is currently beyond the reach of mathematics. We'll describe the classes P and NP using the same informal discussion of algorithms we've been using in this class, to avoid the formal overhead associated with Turing machines — but these classes have mathematically precise definitions that you will have seen in CIS 262 in terms of Turing machines.

Definition 1 *A* problem $R = \{\{I_j\}_{j=1}^{\infty}, V\}$ *is defined by a collection of instances* $I_j \in \{0,1\}^*$ *that may or may not have* solutions $s \in \{0,1\}^*$ *and a verifier* V*. A* verifier *is a mapping* $V : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ *that determines whether* s *is a solution to* I_j*:* $V(I_j, s) = 1$ *"means" that* s *is a solution to* I_j*. The* decision *problem is to determine, given an instance* I_j*, whether there exists a solution* s *such that* $V(I_j, s) = 1$*. The* search *problem is given an instance* I_j *to find an* s *such that* $V(I_j, s) = 1$ *if one exists. We write* $d(I_j) = \mathbb{1}[\exists s : V(I_j, s) = 1]$

A problem R is polynomial time solvable (and said to be in the complexity class P) if there is an algorithm that on every instance I_j of R, runs in time bounded by a polynomial in $|I_j|$, and computes $d(I_j)$.

Definition 2 *P is a set of problems. A problem R is in P if there exists a constant $c > 0$ such that there is an algorithm running in time $O(|I_j|^c)$ that on every instance I_j of R outputs $d(I_j)$.*

Why "polynomial time"? After all we've been working hard in this class to get run times of $O(n^2)$ down to $O(n \log n)$, both of which are "polynomial", and an algorithm running in time $O(n^{100})$ could hardly be called practical. This is motivated by a couple of considerations:

1. **Closure Under Composition**: A polynomial applied to a polynomial yields another polynomial. This is useful for building a clean theory of "efficient algorithms" because it means that we can use polynomial time algorithms as subroutines to build other polynomial time algorithms. For example, if I have a subroutine that runs in time $O(n^2)$, and then I design an algorithm that calls this subroutine $O(n^2)$ times, then my algorithm has running time $O(n^4)$, which is still a polynomial. We will take advantage of this property extensively using our notion of *reductions*. The class of (e.g.) linear time algorithms does not have this nice closure property.

2. **Sharp Distinctions**: The main conjecture to come out of the theory of NP-completeness is that there is a large class of algorithms that *do not even have polynomial time algorithms*. As inefficient as we might think an $O(n^{100})$ time algorithm is, an $\Omega(2^n)$ time algorithm is much worse. The main point of the theory to be developed is the lower bounds, and being permissive about what we call "efficient" means that statements about what is *not* efficient are only stronger.

3. **Practical Experience**: Our experience is that natural problems that we care about — if they are solvable by polynomial time algorithms at all — tend to be solvable by algorithms with low-degree polynomial running times, like $O(n^2)$.

A key idea in the theory of NP-completeness is that of a *polynomial time reduction*

Definition 3 *A problem R is polynomial time reducible to a problem R', written $R \leq_P R'$, if given a polynomial time algorithm $A_{R'}$ for solving problems from R', there is a polynomial time algorithm A_R (which may make calls to $A_{R'}$ as a subroutine) for solving problems from R. In other words, problems in R must be solvable using a polynomial number of standard computational steps* plus *a polynomial number of calls to a subroutine that solves problems in R'.*

We have all used "reductions" informally both in this class and outside of it without defining them, whenever we make calls to subroutines. For example, we showed that the maximum cardinality bipartite matching problem was polynomial time reducible to the maximum flow problem, and we showed that the maximum flow problem was polynomial time reducible to linear program solving.

We make a couple of observations. The first is obvious:

Observation 4 *Suppose $R' \in P$, and $R \leq_P R'$. Then $R \in P$.*

This follows immediately from the definitions and the fact that if $f(\cdot)$ and $g(\cdot)$ are both polynomials, then so is $f(g(\cdot))$.

But it is the *contrapositive* of this statement that will be more important for us:

Claim 5 *Suppose $R \notin P$ — i.e. there is no polynomial time algorithm for R. Then if $R \leq_P R'$, then $R' \notin P$.*

Proof If $R' \in P$ and $R \leq_P R'$, then by the above observation, $R \in P$, a contradiction. ■

This means that polynomial time reductions can be used both to identify a "cluster" of problems that have polynomial time algorithms, and also to identify a "cluster" of problems that *do not* have polynomial time algorithms — at least if we can start with at least one. The theory of NP-completeness develops a large collection of problems that are all polynomial reducible to each other — i.e. such that any of them have polynomial time algorithms if and only if all of them do. We believe none of them do (they have collectively resisted decades of problem solving), but this is currently conjectural, and proving this unconditionally is the $P =? NP$ problem, which is one of the 7 "millenium" problems chosen by the Clay Mathematics institute. Its resolution (in either direction) would entitle you to a $1 million prize.

We also want to be able to speak about problems that are hard for "interesting" reasons. Here is one that is hard for a boring reason: On inputs of length n, output any string s of length $|s| = 2^n$. The difficulty here is just that it would take exponential time to write down the output. Solving this problem is only hard insofar as it would be "hard" to verify that you had solved it by reading the solution. We'll rule out this kind of problem by focusing on problems whose solutions are at least easy to verify. Consider if somebody (possibly untrustworthy) was whispering solutions into your ear during an exam. All you would have to do is to check whether the solution was correct or not. For reasonable problems, this seems much easier than coming up with the solution yourself. And it is the task that defines the class of problems called NP. (NP does *not* stand for "not polynomial" — in fact, every problem in P is also in NP. For historical reasons, NP stands for "non-deterministic polynomial", but never mind this.)

Definition 6 *NP is a set of problems. A problem R is in NP if there exists a constant $c > 0$ such that there is an algorithm running in time $O(|I|^c)$ that on every instance/solution pair (I, s) in R outputs $V(I, s)$.*

Remark Note that we allow the verification algorithm to run in time polynomial only in $|I|$, not $|s|$. This rules out problems that have solutions s with length longer than a polynomial in $|I|$.

Since NP is defined by the verification problem (we are given the solution s), a problem being in NP corresponds to being able to implement the *verifier* in polynomial time, which is a strictly easier problem

than *finding* a solution s in polynomial time. Ok, so lets start out by giving some simple reductions. Here are two different problems that both seem hard (we don't know how to solve either of them in polynomial time):

Definition 7 *An instance of the independent set problem is defined by a graph $G = (V, E)$ and an integer k. An independent set is a subset $S \subseteq V$ such that none of the vertices in S share an edge: for all $u, v \in S, (u, v) \notin E$. A solution to an instance of the independent set problem is an independent set S of size $|S| \geq k$.*

Definition 8 *An instance of the vertex cover problem is defined by a graph $G = (V, E)$ and an integer k. A vertex cover is a subset $S \subseteq V$ that is adjacent to every edge: for every $(u, v) \in E$, either $u \in S$ or $v \in S$. A solution to an instance of the vertex cover problem is a vertex cover S of size $|S| \leq k$.*

Note that both of these problems are in NP, since if we are *given* a purported vertex cover or independent set, it is a simple matter to verify that it is correct. A moment's thought reveals these two problems to in fact be equivalent:

Theorem 9

$$\textit{Independent Set} \leq_P \textit{Vertex Cover, and Vertex Cover} \leq_P \textit{Independent Set}$$

Proof This follows because of the following observation:

Claim 10 *For any graph $G = (V, E)$, S is an independent set if and only if $V \setminus S$ is a vertex cover.*

Proof Consider any set S and let $\bar{S} = V \setminus S$. Suppose S is an independent set. Consider any edge $(u, v) \in E$. It must be that either $u \notin S$ or $v \notin S$. Thus, either $u \in \bar{S}$ or $v \in \bar{S}$, and so $\bar{S}$ is a vertex cover. Conversely, suppose S is a vertex cover. Consider any edge (u, v) and suppose both $u, v \in \bar{S}$. Equivalently, this means that $u, v \notin S$, which would contradict that S is a vertex cover. Thus we have that at most one of u and v are in $\bar{S}$, implying that $\bar{S}$ is an independent set. ■

With this claim in hand, the reductions are simple. Suppose we have a subroutine that can solve Vertex cover problems. Given an instance (G, k) of an independent set problem, we can feed to our vertex-cover subroutine the instance $(G, n - k)$. Similarly, if we have a subroutine for vertex cover, then feeding it $(G, n - k)$ solves the corresponding independent set problem. ■

What we have proved is that *either* both independent set and vertex cover have polynomial time algorithms, or else neither of them do.

So what have we shown so far: If we can solve Independent Set or Vertex Cover in polynomial time, we can solve the other in polynomial time, and if we can solve either of those, we can solve 3-SAT. And all of these problems are clearly *in* NP, since it is easy to verify solutions. We now introduce the idea of "NP-completeness", which identifies, in a sense, a "hard-core" of problems within NP. The NP complete problems are those that have polynomial time solutions if and only if *all* problems in NP have polynomial time algorithms.

Definition 11 *A problem R is NP complete if:*

1. *$R \in NP$, and*
2. *For every problem $X \in NP$, $X \leq_P R$.*

Note that polynomial time reductions compose. So even though its not clear yet how we might show for the first time that a problem is NP complete, if we *already* knew an NP complete problem R, we would have a recipe for showing that a new problem R' was NP complete. We would have to:

1. Show that $R' \in NP$, and

2. For a known NP-complete problem R, show that $R \leq_P R'$.

This would be enough, since if R is NP complete, we already know that for every $X \in NP$, $X \leq_P R$. Hence, because $R \leq_P R'$, we would also have that $X \leq_P R'$.

Both Vertex Cover and Independent Set looked very similar to one another. Here is a problem that looks a little different. To introduce it, we'll need to remember the building blocks of Boolean formulas:

Definition 12 *1. A Boolean variable x_i can take value 0 or 1 (equivalently "True" or "False")*

2. The negation of a Boolean variable $\bar{x}_i = 1 - x_i$. A term *refers to either a Boolean variable or its negation.*

3. A clause is a disjunction of terms:

$$t_1 \vee t_2 \vee \ldots \vee t_\ell = \max(t_1, \ldots, t_\ell)$$

If a clause has ℓ terms in it we say it has length ℓ. A clause is satisfied by assignments to the variables if it takes value 1 *(i.e. if at least one of its terms takes value* 1*).*

4. A collection *of clauses is satisfied by assignments to the variables if they are all satisfied—equivalently, if their conjunction evaluates to 1:*

$$C_1 \wedge \ldots \wedge C_k = \min(C_1, \ldots, C_k).$$

5. An assignment of values to the variables is called a satisfying assignment with respect to clauses $C_1, \ldots, C_k$ it it satisfies all of them. If a set of clauses has a satisfying assignment, we say that they are satisfiable.

Example 1 *Consider the three clauses:*

$$(x_1 \vee \bar{x}_2), (\bar{x}_1 \vee \bar{x}_3), (x_2 \vee \bar{x}_3)$$

The assignment that sets all $x_i = 1$ is not a satisfying assignment because it does not satisfy the 2nd clause. But the assignment that sets all $x_i = 0$ is, so the set of clauses is satisfiable.

Definition 13 *An instance of the satisfiability problem is given by a set of clauses $C_1, \ldots, C_k$ over a set of boolean variables $X = \{x_1, \ldots, x_n\}$. A solution s to an instance of the satisfiability problem is a satisfying assignment. If the instance contains clauses of length* 3*, then it is called an instance of the 3-Satisfiability (3-SAT) problem.*

3-SAT is also a problem in NP: to see this, note that if someone *tells us* a particular assignment to the variables, it is a simple matter to check whether each clause is satisfied or not. But if we aren't given a satisfying assignment, its not clear how to find one faster than exhaustive search. In fact, 3-SAT is a Canonical hard problem, because it straightforwardly represents a set of constraints on a set of decision variables — lots of things can be reduced to 3-SAT. But lets start by observing that 3-SAT is "no harder" than the vertex-cover/independent-set problems we've already seen. This will be an example of a so-called "gadget-reduction": an exercise in "programming" in the language of independent set, using small structured constructions called "gadgets" to represent (in this case) arbitrary 3-SAT instances.

Theorem 14

3-SAT $\leq_P$ Independent Set

Proof We need to show that if we are given a sub-routine for solving independent set problems, we can use it to efficiently solve 3-SAT problems. To do this, lets take an alternative view of 3-SAT. Given an instance $I = \{C_1, \ldots, C_k\}$ of 3-SAT, we need to determine if there is a an assignment to the variables that makes at least one term in each clause evaluate to true. So I has a satisfying assignment if and only if we can pick out one term from each of the k clauses, and assign values to them so that all k terms evaluate to true. Lets show how to take such an instance I, and construct a graph that has an independent set of size k if and only if I is satisfiable.

For each clause $C_j = t_1 \vee t_2 \vee t_3$ in I, we'll construct 3 vertices $v_{j,1}, v_{j,2}, v_{j,3}$, each connected with three edges to form a triangle. So far in the construction, we have k triangles, so the largest independent set has size exactly k: we can select one vertex from each triangle, but we cannot select more than one vertex from the same triangle (because they share an edge).

So far in the construction, however, there is nothing to stop us from selecting a collection of k terms that cannot be simultaniously satisfied: i.e. we could select a pair of vertices $v_{1,1}$ and $v_{2,1}$ representing x_5 and $\bar{x}_5$, which can't be satisfied at the same time. To make sure this can't happen, we also add an edge in our construction between any pair of vertices representing terms that represent a variable together with its negation. A picture is useful here.

We claim that this constructed graph has an independent set of size k if and only if I has a satisfying assignment. First, assume I has a satisfying assignment. In this assignment, there is a(t least one) term in each clause that is satisfied. Pick such a collection of k terms — they form an independent set in our graph. We have picked one vertex from each triangle, so none of them share a triangle edge. And because they are consistent with some satisfying assignment, they do not share any conflict edges either.

In the reverse direction, suppose our graph has an independent set of size k. Because it is an independent set, it must consist of one vertex from each triangle, so we have selected one term from each clause. Because no two vertices share a conflict edge, we can set each term to "true" without any contradiction. No matter how we set the other variable,s this forms a satisfying assignment.

Thus, given a 3-SAT instance, we can solve it by constructing this independent set instance and making a call to an independent set sub-routine. ■

Again, remember what we have shown so far: if we can solve either Independent Set or Vertex Cover in polynomial time, we can solve the other, and if we can solve either of these, then we can solve 3-SAT. All of these problems are clearly in NP, but now we'll give our first NP complete problem, which will ultimately give strong evidence that *none* of these problems can be solved in polynomial time. We won't give the full proof of NP-completeness (which requires a more formal treatment of what an algorithm is, and which you will have already seen in CIS 262), but we'll give the jist of the argument, which is very intuitive.

Definition 15 *A circuit K is a labeled, directed, acyclic graph with the following properties:*

1. *The sources (nodes with indegree 0) are labelled with constants* 0 *or* 1*, or are left as variables x_i: these are the inputs to the circuit.*
2. *Every other node is labelled with a boolean operator from the set $\{\wedge, \vee, \neg\}$. Nodes labelled with $\wedge, \vee$ have indegree 2, and nodes labelled with $\neg$ have indegree 1.*
3. *There is a single node with out-degree 0, representing the output of the circuit.*

A circuit in this sense is transparently modelling a physical circuit, and given an assignment of values to the sources, we evaluate a circuit by sequentially evaluating the Boolean operations labeling its nodes, tracing the evaluation up to the output node. An instance of the problem of circuit-satisfiability is given by a boolean circuit K, and a solution to an instance of the circuit-satisfiability problem is an assignment to the variables that satisfies the circuit — i.e. that causes the output to evaluate to 1.

Theorem 16 (The Cook-Levin Theorem) *The Circuit-Satisfiability problem is NP-complete.*

Proof Sketch Circuit-Satisfiability is clearly in NP, since given a potential satisfying assignment, it is easy to evaluate the circuit in time proportional to its size. The tricky part is to argue that Circuit-Sat $\leq_P R$ for *every* $R \in NP$, without knowing anything about R ahead of time.

The idea is to show that any algorithm can be implemented as a boolean circuit (we won't go into the details, but this is very intuitive, since this is how actual computations are implemented in hardware), and that if the algorithm has worst-case polynomial running time, we only need a circuit of polynomial size to represent it. Once you have convinced yourself of this, fix any problem R in NP, and any problem instance I, and consider the circuit representing its verifier $V(I, \cdot)$, where the problem instance I has been hard-coded in, but the inputs corresponding to a potential solution s are left as variables. Since R is in NP, V can be implemented as a polynomial sized circuit. We can solve the decision problem for I by determining if the polynomially sized circuit representing $V(I, \cdot)$ has any satisfying assignment s. But this is just an instance of the circuit satisfiability problem. So we have shown how to solve an instance of R with a call to a sub-routine for solving Circuit-Satisfiability, which is what we wanted.

Now that we have one NP-complete problem, we can identify more by using polynomial time reductions.

Lemma 17 *If Y is NP-complete, and $X \in NP$ such that $Y \leq_P X$, then X is NP-complete.*

Proof Since $X \in NP$, it only remains to show that for every $R \in NP$, $R \leq_P X$. Since Y is in NP-complete, we know that:

$$R \leq_P Y \leq_P X$$

Since a polynomial of a polynomial is a polynomial, composing the two reductions leads to a polynomial time reduction demonstrating that $R \leq_P X$ as desired. ■

So we now have enough to prove that 3-SAT is NP-complete:

Theorem 18 *3-SAT is NP-complete.*

Proof From the lemma above, it suffices to show that we can efficiently represent Circuit-Satisfiability problems at equivalent 3-SAT instances. The idea, starting with a boolean circuit K, is to represent each node $v \in K$ with a variable x_v such that in any satisfying assignment of the 3-SAT instance we construct, x_v must take the same value as the output of node v in K. First we construct clauses that are only satisfied if the individual gates in the circuit K compute the correct values:

1. If v has label $\neg$, then it has one incoming edge u, and we need to enforce that $x_v = 1 - x_u$. To do this, we construct two clauses: $(x_v \vee x_u)$ and $(\bar{x}_v \vee \bar{x}_u)$. The only way both of these clauses can be satisfied is if exactly one of x_v and x_u evaluate to 1, as desired.

2. If v has label $\vee$, it has two entering edges from u and w. We need to enforce that $x_v = x_u \vee x_w$. To do this we construct clauses $(x_v \vee \bar{x}_u)$, $(x_v \vee \bar{x}_w)$, and $(\bar{x}_v \vee x_u \vee x_w)$.

3. If v has label $\wedge$, it has two incoming edges from u and w. We need to enforce that $x_v = x_u \wedge x_w$. To do this we construct clauses $(\bar{x}_v \vee x_u), (\bar{x}_v \vee x_w)$, and $(x_v \vee \bar{x}_u \vee \bar{x}_w)$.

Finally we need to ensure that vertices labeled as constants take the values they are supposed to. If v is labelled with 1, we add the clause x_v, and if it is labelled as 0 we add the clause $\bar{x}_v$. Finally, we are looking for an assignment to the circuit values that satisfies the circuit, so for the output node o, we add the clause x_o, so any satisfying assignment of our 3-SAT instance must be a satisfying assignment of the circuit K. This completes the reduction. ■

In summary, via reduction, we have shown that not only is Circuit-Satisfiability NP-complete, but so are 3-SAT, Vertex-Cover, and Independent-Set. If we can find polynomial time algorithms for any of these problems, we get polynomial time algorithms for all of them (and all of the other problems in

NP as well!). It turns out that there are thousands of different problems known to be NP-complete, and the more problems we show are NP-complete, the easier it is to expand the set, since to show a new problem to be NP complete, we can reduce from any problem already known to be NP complete. We don't have a proof that $P \neq NP$, but we should view NP-completeness as very strong evidence that a problem does not have a worst-case polynomial time solution.