CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 25

April 26, 2022

Instructor: Sepehr Assadi

1 NP-Hardness Reductions

We continue our study of classes P and NP, and how to show some problems do not admit efficient algorithms (more accurately, at the very least, it is very "hard" to design an efficient algorithm for them).

Equipped with the NP-hardness of Circuit-SAT (namely, Cook-Levin theorem) and the reduction plan outlined in the previous lecture, we are going to prove that a couple of different problems are NP-hard.

1.1 3-SAT Problem

To define the 3-SAT problem, we first need some definition. We say that a boolean formula is a CNF if it is AND of multiple *clauses*, each of which is OR of multiple *literals* (i.e., a variable or its negation). For instance,

- A CNF formula:
$$\underbrace{(a \vee b \vee c)}_{\text{clause}} \wedge (\bar{a} \vee \overbrace{b}^{\text{literal}}) \wedge \underbrace{(\bar{a} \vee \bar{b} \vee c)}_{\text{clause}} \wedge (\overbrace{\bar{b}}^{\text{literal}} \vee \bar{c}).$$
- Not a CNF formula:
$$(a \wedge b \wedge c) \vee (\bar{a} \wedge b).$$

A 3-CNF formula is then any CNF formula where each clause has *at most* 3 literals. We are now ready to define the 3-SAT problem (sometimes called 3-CNF-SAT problem).

Problem 1 (**3-SAT Problem**)**.** We are given a boolean 3-CNF formula Φ with n variables and m clauses. For any $y \in \{0,1\}^n$, we use $\Phi(y)$ to denote the value of the formula when assigning y to the variables of the formula. The 3-SAT problem asks does there exists at least one assignment y such that $\Phi(y) = 1$ or not? In other words, is this 3-CNF formula *ever* satisfiable or not?

3-SAT is in NP. Similar to the Circuit-SAT problem a simple verifier is as follows:

- Proof: If Φ is satisfiable then there should exists some assignment y to the variables where $\Phi(y) = 1$. We can use any such y as a proof.
- The verifiers then gets the input formula Φ and the proof y that $\Phi(y) = 1$. We can evaluate $\Phi(y)$ by computing the value of each clause and checking whether every clause is satisfied or not and hence check for ourself whether $\Phi(y) = 1$ indeed or not.

As we designed a poly-time verifier for this problem, this means that 3-SAT is in NP.

3-SAT is NP-hard. The goal is to show that *any* polynomial time algorithm for 3-SAT also implies a polynomial time algorithm for Circuit-SAT (and since Circuit-SAT is NP-hard, this implies P = NP, which in turn implies 3-SAT is also NP-hard). The reduction is as follows.

Reduction: Given an input C to the Circuit-SAT problem, we turn it into a 3-CNF formula Φ as follows:

1. Define a new variable for every *wire* in the circuit C (including the input wires called variables and the output wire). These are all the variables in the formula Φ.
2. For every gate G in the circuit C we add the following clauses to the formula:
   (a) AND: Let a be the variable for the output wire of this gate and b and c be the variables for input wires. We want to have $a = b \wedge c$ in our formula. To do so, we add the following clauses to Φ:
   $$(a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c).$$
   (b) OR: Let a be the variable for the output wire of this gate and b and c be the variables for input wires. We want to have $a = b \vee c$ in our formula. To do so, we add the following clauses to Φ:
   $$(\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}).$$
   (c) NOT: Let a be the variable for the output wire of this gate and b be the variable for the input wire. We want to have $a = \bar{b}$ in our formula. To do so, we add the following clauses to Φ:
   $$(a \vee b) \wedge (\bar{a} \vee \bar{b}).$$
3. This concludes the description of the formula Φ (we emphasize that for the *output* gate, if z is variable for the output wire, we add the singleton clause z to Φ as well).

We then simply run our (supposed) algorithm for 3-SAT on the formula Φ and if the algorithm outputs Φ is satisfiable, we output that C is satisfiable and otherwise output C is not satisfiable.

Proof of Correctness: Let C be any given circuit and Φ be the resulting 3-CNF formula in the reduction. We prove that C is satisfiable if and only if Φ is satisfiable. This will immediately imply the correctness.

In order to do this, we first prove that Φ and C are *logically equivalent*. To do this, we need to show that the set of clauses introduced for each gate work exactly the same as the gate itself:

- AND-gates – By writing the truth table of each of the two expressions we have:

$a = b \wedge c$

a	b	c	True/False
0	0	0	True
0	0	1	True
0	1	0	True
0	1	1	False
1	0	0	False
1	0	1	False
1	1	0	False
1	1	1	True

$(a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c)$

a	b	c	True/False
0	0	0	True
0	0	1	True
0	1	0	True
0	1	1	False
1	0	0	False
1	0	1	False
1	1	0	False
1	1	1	True

  So AND-gates and the resulting clauses in the reductions are equivalent.

- OR-gates – By writing the truth table of each of the two expressions we have:

$a = b \vee c$

a	b	c	True/False
0	0	0	True
0	0	1	False
0	1	0	False
0	1	1	False
1	0	0	False
1	0	1	True
1	1	0	True
1	1	1	True

$(\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c})$

a	b	c	True/False
0	0	0	True
0	0	1	False
0	1	0	False
0	1	1	False
1	0	0	False
1	0	1	True
1	1	0	True
1	1	1	True

  So OR-gates and the resulting clauses in the reductions are equivalent.

- NOT-gates – By writing the truth table of each of the two expressions we have:

$a = \bar{b}$

a	b	True/False
0	0	False
0	1	True
1	0	True
1	1	False

$(a \vee b) \wedge (\bar{a} \vee \bar{b})$

a	b	True/False
0	0	False
0	1	True
1	0	True
1	1	False

  So NOT-gates and the resulting clauses in the reductions are equivalent.

We can now use this to finalize the proof. Recall that we need to prove C is satisfiable if and only if Φ is satisfiable. As any other "if and only if" statement, we need to prove this in two steps:

(i) If C is satisfiable, then Φ is satisfiable also: Pick a satisfying assignment x to C and consider every wire of this circuit. Let y be the assignment of these wires to variables in Φ. By the above part, C and Φ are logically equivalent and thus y satisfies every clause in Φ. Finally, since for the output wire we have a singleton clause z (which is equivalent to $C(x)$ which in turn is 1), the $\Phi(y) = 1$. This means Φ is satisfiable.

(ii) If Φ is satisfiable, then C is satisfiable also. Pick a satisfying assignment y to Φ and consider the variables assigned to the *input* wires. Let x be the assignment of these input wires in C. By the above part, C and Φ are logically equivalent and so every wire of circuit C on the input x will get the same value as the corresponding variable y in Φ. This in particular means that the output wire gets the value 1 on the input x (because the output wire is a singleton clause) and thus $C(x) = 1$. This means C is satisfiable.

Runtime analysis: We also need to show that *if* we have a poly-time algorithm for 3-SAT the above reduction runs in poly-time. This is true because the size of Φ is at most a constant factor larger than size of the circuit (each gate is replaced by at most 3 clauses) and we create Φ in linear-time from C. Thus a poly-time algorithm for 3-SAT on Φ implies a poly-time algorithm for Circuit-SAT on Φ.

This concludes the proof of NP-hardness of the 3-SAT problem. Given that we already proved 3-SAT is also in NP, this means 3-SAT is NP-complete.

1.2 Maximum Independent Set Problem

Unlike Circuit-SAT which might be a bit cumbersome to work with, 3-SAT is an excellent problem for doing reductions from. We now use this to prove NP-hardness of a fundamental graph problem, namely, the *maximum independent set* problem.

Given an undirected graph $G = (V, E)$, a set $S \subseteq V$ of vertices is called an *independent set* if there is *no* edge with both endpoints in S (in other words, no vertices in S are neighbor to each other).

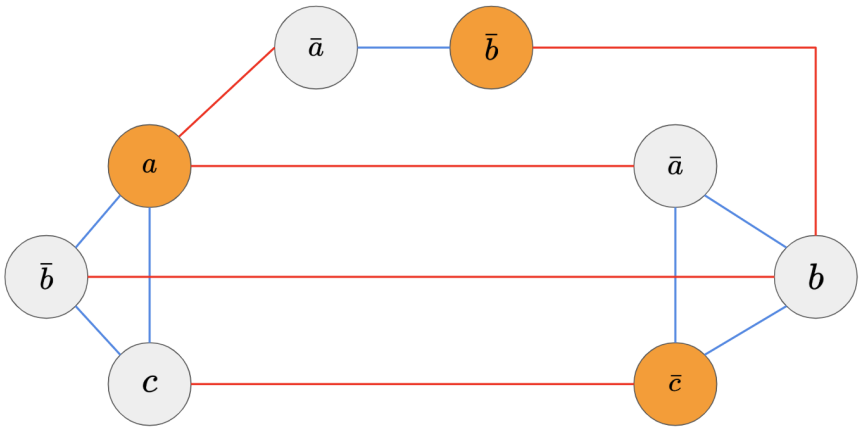
Problem 2 (**Maximum Independent Set Problem**)**.** Given an undirected graph $G = (V, E)$ output the size of the largest independent set in G. We abbreviate this problem by ***MaxIndSet***.

Is MaxIndSet in NP? MaxIndSet is *not* a decision problem and hence cannot be in NP.

MaxIndSet is NP-hard. Nevertheless, we prove that MaxIndSet is NP-hard, meaning that a poly-time algorithm for MaxIndSet implies P = NP. We do this by a reduction from 3-SAT (or alternatively, show that 3-SAT can be reduced to MaxIndSet).

Reduction: Given any instance Φ of the 3-SAT problem, we create an instance $G(V, E)$ of maximum independent set as follows:

1. For any clause in Φ, we add (at most) three vertices corresponding to the literals of the clause to V.
2. Two vertices in G are connected to each other if either (1) they corresponds to the variables in the same clause (and were added to G together in the last step), or (2) they correspond to a variable and its negation. See Figure 1 for an example.



$$(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b}) \wedge (\bar{a} \vee b \vee \bar{c})$$

Figure 1: An illustration of the reduction of 3-SAT to MaxIndSet (here blue edges connect vertices in the same clause and red edges connect a vertex to its negation). The orange vertices form an independent set in G corresponding to the assignment of the variables $a = 1$, $b = 0$, and $c = 0$.

We then simply run our (supposed) algorithm for MaxIndSet on G and let o denote size of maximum independent in G output by the algorithm. If o is equal to to the number of clauses in Φ, we output Φ is satisfiable and otherwise we output it is not.

Proof of Correctness: Let k denote the number of clauses in Φ. We first claim that any independent set in G has size *at most* k. This is because all vertices inside a clause are connected to each other and hence any independent set can pick at most one variable from a clause. Now to prove the correctness of the reduction (similar to any other reduction), we should prove that the size of maximum independent set in G is *exactly* k if and only if Φ is satisfiable.

(i) Maximum independent set size in G is $k \implies \Phi$ is satisfiable: Let $S = \{v_1, \ldots, v_k\}$ be an independent set of size k in G. Note that (1) each vertex belongs to a unique clause (because vertices in each clause

are connected to each other and hence cannot be part of an independent set together), and (2) a vertex and its negation do not appear *simultaneously* in S (because each variable-vertex is connected to its negation). We can thus make the *literals* corresponding to $v_1, \ldots, v_k$ in Φ to be true in Φ (and pick an arbitrary assignment of remaining variables) and satisfy all clauses in Φ. This means that Φ is satisfiable.

(ii) Φ is satisfiable $\implies$ maximum independent size in G is k: Let y be a satisfying assignment of Φ. We pick a set S of vertices in G by picking one variable-vertex from each clause corresponding to a true literal in y (since y is a satisfying assignment, there exists always one such literal and if there is more than one we can pick arbitrarily). Since in any assignment of y, we will never have a literal and its negation to be true simultaneously and since we are only picking exactly one literal per each clause, the set S is an independent set of size k. Finally since any independent set has size at most k in G, we get the result.

Runtime analysis: We can create the graph G in polynomial time from Φ and *if* we have a poly-time algorithm for MaxIndSet, we will obtain a poly-time algorithm for 3-SAT as well.

This concludes the proof of NP-hardness of MaxIndSet problem since we proved a poly-time algorithm for MaxIndSet implies a poly-time algorithm for an NP-hard problem, namely, 3-SAT. [1]

1.3 Minimum Vertex Cover Problem

Let us now study another reduction. Given an undirected graph $G = (V, E)$, a set $T \subseteq V$ of vertices is called a *vertex cover* if every edge of G has at least one endpoint in T; informally speaking, vertices of T are "covering" all edges of G (hence the term vertex cover).

Problem 3 (**Minimum Vertex Cover Problem**)**.** Given an undirected graph $G = (V, E)$ output the size of the smallest vertex in G. We abbreviate this problem by ***MinVC***.

Is MinVC in NP? MinVC is *not* a decision problem and hence cannot be in NP.

MinVC is NP-hard. Nevertheless, we prove that MinVC is NP-hard, meaning that a poly-time algorithm for MinVC implies P = NP. We do this by a reduction from MaxIndSet (or alternatively, show that MaxIndSet can be reduced to MinVC).

Reduction: Suppose we have an algorithm A for the MinVC problem. Then, given any instance G of the MaxIndSet problem, we simply run A on G to find a minimum vertex cover T of G and then return $n - |T|$ as the answer to MaxIndSet Problem.

Proof of Correctness: To prove the correctness of the reduction (similar to any other reduction), we should prove that there is a vertex cover of size k, for any k, in G if and only if there is an independent set of size $n - k$ in G; this then implies that finding minimum vertex cover size also finds the maximum independent set (minimizing k results in maximizing $n - k$) and thus the answer is correct.

(i) A vertex cover of size k $\implies$ an independent set of size $n - k$:

Let T be the vertex cover of size k. Then, consider the set $S = V - T$ which has size $n - k$. We claim that S is an independent set: Suppose not; then there is an edge (u, v) with both $u, v \in S$ and so neither of u and v are in T; this means the edge (u, v) is not covered by T, a contradiction with T being a vertex cover. So G has an independent set S of size $n - k$ in this case.

[1] We can trace the chain: poly-time algorithm for MaxIndSet $\underbrace{\implies}_{this\ reduction}$ poly-time algorithm for 3-SAT $\underbrace{\implies}_{previous\ reduction}$ poly-time algorithm for Circuit-SAT $\underbrace{\implies}_{Cook-Levin\ Theorem}$ poly-time algorithm for all NP problems, i.e., P=NP.

(ii) An independent set of size $n-k \implies$ a vertex cover of size k:

Let S be the independent set of size $n-k$. Then, consider the set $T = V - S$ which has size k. We claim that T is a vertex cover: Suppose not; then there is any edge (u, v) not covered by T which means neither of u and v are in T; this means both $u, v \in S$ and so there is an edge with both endpoints in S, a contradiction with S being an independent set. So G has a vertex cover T of size k in this case.

Runtime analysis: If we have a poly-time algorithm for MinVC, we will obtain a poly-time algorithm for MaxIndSet as well.

This concludes the proof of NP-hardness of MinVC problem since we proved a poly-time algorithm for MinVC implies a poly-time algorithm for an NP-hard problem, namely, MaxIndSet problem.[2]

2 Concluding Remarks on NP-Hardness Reductions

The general strategy of proving a problem NP-hard in this course is always the same as the three different reductions we already saw: to show problem A is NP-hard, we will find another NP-hard problem B and show that solving B in polynomial time reduces to solving A in polynomial time; this implies that a polynomial time algorithm for A will give a polynomial time algorithm for B which in turn, by definition, implies that P = NP. [3]

Each reduction also works as follows (more or less): give an instance problem B, we should create another instance of problem A such that the answer to problem A uniquely determines the answer to problem B. Finally, for the proof of correctness, we should use the same strategy as any other reduction: the answer to problem A in this new instance is "something" if and only if the answer to problem B in the given instance is "something". We should also not forget to prove that the reduction itself takes polynomial time and thus a poly-time algorithm for A indeed gives a poly-time algorithm for B[4].

In the following, we list several other "(relatively) easy-to-show NP-hard" problems that appear very frequently, and are worth knowing.

- **Maximum Clique Problem.** A clique in an undirected graph $G = (V, E)$ is any set T of vertices such that there is an edge between any pairs of vertices.

  The maximum clique problem asks for finding the size of the largest clique in a given graph. Maximum clique problem is an NP-hard problem and the easiest way to prove this is to do a reduction from the maximum independent set problem (this is something quite easy at this point and you are encouraged to do this on your own for practice).

- **3-Coloring Problem.** A 3-coloring of an undirected graph $G = (V, E)$ is a function $C : V \to \{1, 2, 3\}$ that assigns one of the colors $\{1, 2, 3\}$ to vertices of the graph so that every edge of the graph has two different colors at its end point, i.e., $C(u) \neq C(v)$ for every edge $\{u, v\} \in E$.

  The 3-coloring problem asks whether a given undirected graph admits a 3-coloring or not (this is a decision problem). One way to prove 3-coloring is NP-hard is by a reduction from the 3-SAT problem; see Chapter 12.10 of Erickson's book (this is a nice reduction and you are encouraged to check it).

[2] We can trace the chain: poly-time algorithm for MinVC $\underbrace{\implies}_{\text{this reduction}}$ poly-time algorithm for *MaxIndSet* $\underbrace{\implies}_{\text{reduction of previous section}}$ poly-time algorithm for 3-SAT $\underbrace{\implies}_{\text{previous reduction}}$ poly-time algorithm for Circuit-SAT $\underbrace{\implies}_{\text{Cook–Levin Theorem}}$ poly-time algorithm for all NP problems, i.e., P=NP.

[3] Note that this task involves two different steps: (1) finding the appropriate problem B in the first place, and (2) performing the reduction. However, in this course, you will *typically* be told which problem you should reduce from, namely, the choice of B, or at least a small set of candidates for B, will be given to you.

[4] If the reduction is *not* poly-time, there is no reason a poly-time algorithm for A give a poly-time algorithm for B, no?

- **Hamiltonian Cycle Problem.** A Hamiltonian cycle in an undirected graph $G = (V, E)$ is a cycle that passes through *every* vertex.

  The Hamiltonian cycle problem asks whether a given undirected graph has any Hamiltonian cycle or not (this is a decision problem). Hamiltonian cycle can also be shown to be NP-hard by a reduction from the 3-SAT problem; see Chapter 12.11 of Erickson's book. This problem has several other well-known NP-hard variants such as finding a Hamiltonian path instead (a path that goes through every vertex) or the traveling salesman problem (TSP).

- **Minimum Set Cover Problem.** A set cover of a collection sets $S_1, \ldots, S_m$ where each $S_i \subseteq \{1, \ldots, n\}$ is any collection of sets C such that every element in $\{1, \ldots, n\}$ belongs to at least one of the sets in C (namely, every element is covered by C) or in other words $\bigcup_{i \in C} S_i = \{1, \ldots, n\}$.

  The minimum set cover problem asks for finding the size of the smallest set cover of a given collection of sets. Set cover can be proven NP-hard by a reduction from the minimum vertex cover problem (this is a relatively easy reduction and it would be a good idea to think about it on your own – simply try to find the (rather direct) connection between set cover and minimum vertex cover). Set cover problem also has several other well-known NP-hard variants such as the hitting set problem or the dominating set problem.

- **Knapsack Problem.** We have already seen this problem in details when studying dynamic programming algorithms. Knapsack problem is also another NP-hard problem and can be proven so by a reduction from 3-SAT. It is important to emphasize again that the solutions we designed for this problem using dynamic programming were *not* polynomial time[5]. Knapsack problem also has several other well-known NP-hard variants such as the Partition problem and the Subset Sum problem.

The above list is by no means a comprehensive list of NP-hard problems (not even the most interesting ones). However, it will hopefully give you a general sense of various NP-hard problems.

[5]This means that quite unfortunately we have not proved P=NP ...