CS 31: Algorithms (Spring 2019): Lecture 19

Date: 28th May, 2019

Topic: P, NP, and all that Jazz.

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

The main goal of this lecture is to introduce the notions of $\mathbf{NP}$ and $\mathbf{NP}$-hardness and $\mathbf{NP}$-completeness, and to understand what the $\mathbf{P}$ vs $\mathbf{NP}$ question is. Our treatment would be semi-formal; for a formal and much more in-depth treatment I suggest taking CS 39. We start with *decision problems*.

1 Decision Problems

Definition 1. Informally, any problem Π is a decision problem if its solutions are either YES or NO. Formally, a problem Π is a decision problem if all instances $\mathcal{I} \in \Pi$ can be *partitioned* into two classes YES-Instances and NO-instances, and given an instance $\mathcal{I}$, the objective is to figure out which class it lies in.

Example 1. We have seen many examples of decision problems in this course.

1. (Subset Sum) Given $a_1, \ldots, a_n; B$ decided whether or not there exists a subset of these numbers which sum to exactly B?
2. (Reachability) Given a graph $G = (V, E)$ and two vertices s, t, is there a path from s to t in G?
3. (Strongly Connected) Given a directed graph $G = (V, E)$, is it strongly connected?

Not all problems are decision problems. There are *optimization* problems which ask us to maximize or minimize things. For example, find the largest independent set in a graph; given an array, find the maximum range sub-array; given a knapsack instance, find the largest profit subset that fits in the knapsack. Most optimization problems, however, have *decision* versions. Let's give some examples.

Example 2.
1. Given a graph and a number K, is there an independent set in G of size $\geq K$?
2. Given an array $A[1:n]$ and a number K, are there $i < j$ such that $A[j] - A[i] \geq K$?
3. Given a knapsack instance and a number K, is there a subset of items which fit in the knapsack and give profit $\geq K$?

You get the picture.

Now that we have defined decision problems, let's move to *polynomial time reductions*.

2 Polynomial Time Reductions

Definition 2. A decision problem Π_A **polynomial time reduces** to a decision problem Π_B if there exists an algorithm $\mathcal{A}$ such that

- $\mathcal{A}$ takes input an *instance* $\mathcal{I}$ of Π_A.
- $\mathcal{A}$ outputs an *instance* $\mathcal{J}$ of Π_B.
- There exists a polynomial $p(\cdot)$ such that the running time of $\mathcal{A}$ on $\mathcal{I} \in \Pi_A$ is at most $p(|\mathcal{I}|)$ where $|\mathcal{I}|$ is the size of the instance. This necessarily implies $|\mathcal{J}| \leq p(|\mathcal{I}|)$.
- $\mathcal{I}$ is a YES-instance of Π_A **if and only if** $\mathcal{J}$ is a YES-instance of Π_B.

In that case, we say $\Pi_A \preceq_{\text{poly}} \Pi_B$, or simply, $\Pi_A \preceq \Pi_B$.

Indeed, we have already seen, and you have written about, reductions. In each application of the maximum flow, we reduced the application to maximum flow. For example, to check if a bipartite graph has a matching of size k, we saw an *algorithm* which takes an instance of bipartite matching (a bipartite graph G) and constructs a network (N_G) with the property that if the graph has a matching of size k, then the max-flow in N_G is k, and *also* vice-versa, namely, if N_G has a max-flow of k, then G has a matching of size k.

Why are reductions useful? Because they allow us to state in a formal sense when a problem Π_A is "easier" than Π_B. Suppose we had a polynomial time algorithm $\mathcal{B}$ to solve problem Π_B, and if $\Pi_A \preceq \Pi_B$, then we can also solve Π_A in polynomial time. To see this, note that given any instance $\mathcal{I}$ of Π_A, we can get $\mathcal{J} = \mathcal{A}(\mathcal{I})$ in polynomial time, and then use the algorithm $\mathcal{B}$ to decide whethet $\mathcal{J}$ is a YES instance or not. Whatever it is, we can give the same answer for $\mathcal{I}$. Let's encapsulate this in a lemma.

Lemma 1. If $\Pi_A \preceq \Pi_B$, and there exists a polynomial time algorithm to solve Π_B, then there is a polynomial time algorithm to solve Π_A.

The other nice thing is transitivity.

Lemma 2. If $\Pi_A \preceq \Pi_B$ and $\Pi_B \preceq \Pi_C$, then $\Pi_A \preceq \Pi_C$.

✍

Exercise: *Prove this.*

So where are we getting at? Suppose we are stuck at solving a problem Π_B. We try and try but never get an efficient algorithm. What should we do? Suppose there was some other *famous* problem Π_A which many more people have looked at, many more people have tried, and many more people have failed at getting polynomial time algorithms. And suppose you can now *prove* $\Pi_A \preceq \Pi_B$. Well, that will give very good indication that Π_B is a hard nut to crack; by Lemma 1 solving Π_B will amount to solving Π_A in polynomial time as well. Next, we describe this famous "hard" Π_A.

3 A "Hard" Problem

The problem is a cousin of a problem you already met in the problem sets. Recall what Boolean formulas are. There are Boolean variables and their negations, together called *literals*. Examples are x_1, $\overline{x}_2$, and so on. Then there are *clauses* where each clause is a collection of literals orred with each other. Examples being $x_1 \vee \overline{x}_2$, $x_2 \vee x_3 \vee \overline{x}_4$, and so on. A *SAT formula* is a collection of various clauses. A formula is *satisfiable* if there exists a setting of $\{$true, false$\}$ to the variables such that in *every* clause *at least* one of the literals evaluates to true. For example, the formula could be

$$\phi = (x_1 \vee \overline{x}_2) \ \wedge \ (x_2 \vee \overline{x}_1)$$

in which case it is satisfiable by setting both x_1 and x_2 to true. On the other hand, the formula

$$\phi' = (x_1 \vee \overline{x}_2) \ \wedge \ (x_2 \vee \overline{x}_1) \ \wedge \ (\overline{x}_1 \vee \overline{x}_2) \ \wedge \ (x_1 \vee x_2)$$

is *unsatisfiable*. You can check this by going over all the 4 possibilities of setting $\{$true,false$\}$ to the x_1, x_2 variables.

SAT
Input: A SAT formula with n variables and m clauses.
Output: Decide whether or not it is satisfiable.

SAT is a decision problem. We don't know of any polynomial time algorithm for this problem.

Conjecture 1. There is no polynomial time algorithm for SAT.

Remark: *At this point, you may be inclined to say, "Sure! This problem is so unstructured, what can you do but check all assignments? And there are 2^n many of them." Well, don't! First, there are many heuristics which do better than pure guessing and do well in practice. Unfortunately, none of them provably give fast algorithms all the time. Second, the argument above can be made for many problems. And if something we have learned in this course is that the study of algorithms is riddled with surprises (and we are surprised by many riddles).*

The above is a conjecture. We are very far from either proving or disproving this conjecture. On the one hand, the best algorithm for SAT runs in time "essentially" 2^n (formally, it runs in $2^{n-o(n)}$ time where recall $o(n)$ is the set of functions whose ratio with n tend to 0 as n tends to infinity). On the other hand, $O(n)$ algorithms for SAT are not ruled out. On a different track, as mentioned above, there are many fast heuristics for SAT which solve the SAT formulas arising in practice very well. All in all, SAT is a very interesting beast which hasn't been tamed.

Here is another simpler sounding version of SAT.

3SAT
Input: A SAT formula with n variables, m clauses, and at most 3 literals per clause
Output: Decide whether or not it is satisfiable.

The next lemma shows that 3SAT is no simpler – in fact SAT can be reduced to 3SAT.

Lemma 3. SAT $\preceq$ 3SAT

Proof. Let's recall what we need to do. We need to find an algorithm which takes an instance of SAT, that is a formula, and returns an instance of 3SAT such that the SAT formula is satisfiable if and only if the 3SAT formula is.

Let ϕ be the SAT formula. Of course if all clauses had at most 3 literals, there was nothing to do. So let us consider a clause C with more than 3 variables. Say, $C = (x_1 \vee \overline{x}_2 \vee x_3 \vee x_4)$ with 4 literals. Our reduction algorithm *introduces a new variable* y_C and breaks C into 2 clauses as follows:

$$(x_1 \vee \overline{x}_2 \vee y_C) \quad \wedge \quad (\overline{y_C} \vee x_3 \vee x_4)$$

We claim that C is satisfiable if and only if the above two clauses are. If C is satisfiable, then one of its literal is set to true. This literal appears in one of the two clauses above, say the first one. Then we set $\overline{y_C}$ to true making the second clause satisfied as well. On the other hand, if the above two clauses are satisfiable, then exactly y_C or $\overline{y_C}$ is set to true. If the former, then one of the literals in the second clause which belongs to C must be set to true. Which means C is also true.

Therefore, we can replace C by the above two clauses, which as you can see has 3 literals each. More generally, if C has ℓ literals we could replace it with two clauses one of which has 3 literals and the other has $\ell - 1$ literals. We can then continue chipping on the $(\ell - 1)$ length clause. Each time we would introduce a new variable and a new clause. Therefore, a clause C of length ℓ can be transformed to an equivalent 3SAT formula with $\ell - 3$ extra variables and clauses such that the original clause is satisfiable if and only if the 3SAT formula is.

We can repeat the above for every clause. The resulting 3SAT formula will have at most $O(nm)$ variables and $O(nm)$ clauses and has the property that it is satisfiable if and only if the original SAT formula is. Also note that the above reduction time is at most a polynomial of $(n + m)$, in fact, the total time is $O((n + m)^2)$. This completes the proof. $\square$

Remark: *You should observe that the number 3 can't be brought down to 2 using the above trick. If we try to break a clause of size 3 into two clauses with an extra variable, we end up with clauses of size 3. In fact this shouldn't be a surprise; you have seen a linear time algorithm to solve* 2SAT.

✍

Exercise: *Show a reduction from* 3SAT *to* EXACT3SAT *where every clause of the satisfiability formula has exactly* 3 *literals.*

Ok, so the general satisfiability formula can be reduced to another version with some extra structure. Why is it useful for anyone who say works on graph algorithms or something else, and doesn't care much about satisfiability? The next lemma shows how reductions can straddle domains. A little later we will see that SAT is the mother of "most" computation problems.

Consider the following problem. Call a subset $I \subseteq V$ of vertices in a graph *independent* if for every pair of vertices u and v in I, (u, v) is ***not*** and edge.

IS
Input: An undirected graph $G = (V, E)$ and a parameter K.
Output: Decide whether or not there exists an independent set I in G with $|I| \geq K$?

Lemma 4. 3SAT $\preceq$ IS

Proof. Again, let's recall what we need to do. We need to come up with an algorithm which takes a 3SAT formula and returns a graph G and a parameter K such that (a) if the formula is satisfiable then G has an independent set of size $\geq K$, and (b) if the formula is not satisfiable, then G can't have an independent set of size $\geq K$. It is often easier to show the contrapositive of (b), that is, (b') if G has an independent set of size $\geq K$, then the formula is satisfiable.

For simplicity, and also because of the exercise above, we work with EXACT3SAT where each clause has exactly 3 literals. The reduction is as follows. The graph has $2n+3m$ vertices. These are divided into two classes: V_{var} and V_{cl}. For each variable x_i of the formula ϕ, we add vertex x_i and $\overline{x}_i$ in V_{var}. We add an edge between them. For each clause $C = (\alpha \vee \beta \vee \gamma)$, we add three vertices (α, β, γ) to V_{cl}. We add all three edges between them.

Now note that for the same literal, say x_1 there is exactly one appearance in V_{var}, but there can be many appearances in V_{cl}. In fact it appears as many times this literal appears in the formula. Next comes the crucial *connector* edges. For every literal α, we add an edge between the unique version in V_{var} to *every* appearance of the literal α in V_{cl}.

We give an example in Figure 1. Perhaps you should try out one for yourself too.
To complete the reduction we also need to specify the parameter K. We set $K = n + m$.

Now that the reduction is defined, we need to prove that it does what it is supposed to do.

- Suppose ϕ is satisfiable. We now construct an independent set I of size $n + m$. For every α that is set to true, we pick the vertex corresponding to the *complement* $\overline{\alpha}$ from V_{var} in I. At this point note I is independent and is of size n. For every clause $C = (\beta \vee \gamma \vee \delta)$ we know *at least* one of the literals must be set to true. We *arbitrarily* choose one of them, say β, and pick the corresponding vertex in V_{cl} for this clause

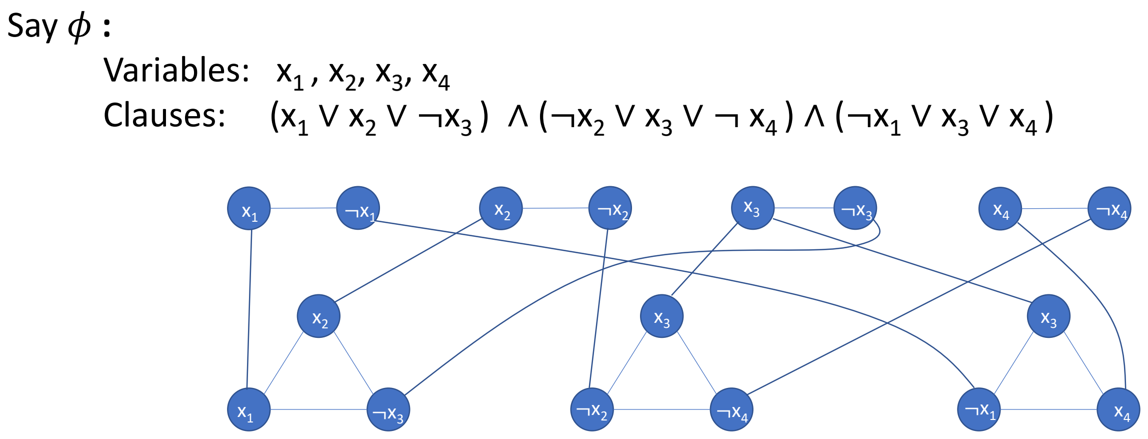
Say ϕ :
Variables: x_1, x_2, x_3, x_4
Clauses: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_3 \vee x_4)$
x_1
$\neg x_1$
x_2
$\neg x_2$
x_3
$\neg x_3$
x_4
$\neg x_4$
x_2
x_3
x_3
x_1
$\neg x_3$
$\neg x_2$
$\neg x_4$
$\neg x_1$
x_4

Figure 1: Example for the reduction 3SAT $\preceq$ IS

into I. Since β is set to true, we *don't* pick the vertex corresponding to β in V_{var}. And thus I remains independent. Since across clauses vertices from V_{cl} don't have edges, we see that I remains independent after we pick one vertex from V_{cl} for each clause. Thus we get I of size $(n + m)$.

- Now we need to prove that if there is an independent set I of size $\geq n + m$, then the formula is satisfiable. Firstly observe that G contains n disjoint edges and m disjoint triangles. Any independent set can select at most one vertex from each such edge and each such triangle. Thus, if $|I| \geq n + m$, then in fact $|I| = n + m$ and must pick one vertex from each edge and one vertex from each triangle.

  Suppose we pick vertex $\alpha \in V_{\text{var}}$ from the edge $(\alpha, \overline{\alpha})$. Then set α to *false*. This defines an assignment to all variables. We claim this satisfies the formula. Pick a clause $(\alpha \vee \beta \vee \gamma)$. We know that among the corresponding vertices in V_{cl} we pick exactly one, say γ. But this means that among the two vertices $(\gamma, \overline{\gamma})$ in V_{var} we must have $\overline{\gamma}$ in the independent set. That is, we must have set γ to true thus satisfying the clause. Done.

You should "run" this proof on the example in Figure 1 to make sure you see what's going on. $\square$

Let's see what conclusions we can make at this point. Lemma 2 applied to Lemma 3 and Lemma 4 implies that SAT $\preceq$ IS. Thus if you believe that the completely unstructured problem SAT is "hard", that is if you believe Conjecture 1, then you get the conclusion "there are no polynomial time algorithms for IS" using Lemma 1.

4 P vs NP

We finally come to the main goal of this lecture: after this whenever you use the jargon "**NP**-hard" or "**NP**-complete", you will *know* what you mean by it, and *never* say things like

"You know, the problem's like NP-hard so yeah not polynomial time."

Before I begin, I confess we have done a bit of disservice to the concepts I am going to tell about by choosing horrible names like **P** and **NP**. But it is what it is and we are stuck with it for better or worse. Once again, let me remind you, what I am saying below is *informal*. It is not incorrect, but please take CS39 to get a better appreciation of all this.

Recall decision problems. **P** and **NP** are just classes (subsets) of decision problems. The definition of **P** is straightforward.

Definition 3. **P** is the class of decision problems which can be solved in polynomial time.
More precisely, $\Pi \in \mathbf{P}$ if there exists an algorithm $\mathcal{A}$ and a polynomial $p()$, such that

- For every YES-instance $\mathcal{I} \in \Pi$, $\mathcal{A}(\mathcal{I})$ runs in time $p(|\mathcal{I}|)$ and returns YES.
- For every NO-instance $\mathcal{I} \in \Pi$, $\mathcal{A}(\mathcal{I})$ runs in time $p(|\mathcal{I}|)$ and returns NO.

The definition of **NP** is more interesting. **NP** is the class of decision problems which can be *verified* in polynomial time. What? Let us elaborate in the definition below.

Definition 4. A problem Π lies in **NP** if there exists an algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$, such that

- For every YES-instance $\mathcal{I} \in \Pi$, there *exists* a "solution" $\mathcal{S}$ with $|\mathcal{S}| \leq p(|\mathcal{I}|)$ such that $\mathcal{A}(\mathcal{I}, \mathcal{S})$ runs in $\leq p(|\mathcal{I}| + |\mathcal{S}|)$ time and returns YES.
- For every NO-instance $\mathcal{I} \in \Pi$, and for *every* purported "solution" $\mathcal{S}$ of size $|\mathcal{S}| \leq p(|\mathcal{I}|)$, $\mathcal{A}(\mathcal{I}, \mathcal{S})$ runs in $\leq p(|\mathcal{I}| + |\mathcal{S}|)$ time and returns NO.

The above definition is deep, so let's first look at an example and then we see other ways of interpreting it.

Claim 1. SAT lies in the class **NP**.

Proof. We need to come up with an algorithm $\mathcal{A}$ which reads an instance of SAT, that is a formula ϕ, and a "solution" $\mathcal{S}$ and says YES or NO. Here is one observation: we can assume that the solution $\mathcal{S}$ be any given "format"; our algorithm $\mathcal{A}$ will say NO immediately if the format is not satisfied.

So here is what our "verifier" $\mathcal{A}$ expects: $\mathcal{I}$ should be a formula ϕ, and the solution $\mathcal{S}$ should be an assignment of truth values to the variables of ϕ. Anything else, $\mathcal{A}$ will reject outright (that is, says NO). Furthermore, if the format is correct, the algorithm just verifies if the truth value expressed in $\mathcal{S}$ satisfies the formula ϕ. If so, it says YES.

Now we see that the conditions of the definition are trivially true. If ϕ is indeed satisfiable, then $\mathcal{S}$ be the assignment which satisfies ϕ. We can't stress enough: the algorithm $\mathcal{A}$ doesn't need to *find* $\mathcal{S}$; it just needs to *verify* it. A much easier proposition. In any case, if ϕ is satisfiable then the $\mathcal{S}$ described would make the algorithm $\mathcal{A}$ accept. All this takes linear time (so the polynomial is just a linear function).

On the other hand if the formula is not satisfiable, then if no assignment makes all clauses true. So if $\mathcal{S}$ is not rejected outright, it will be rejected once all the clauses are verified by $\mathcal{A}$. That is, if $\mathcal{I}$ is a NO-instance, no matter what $\mathcal{S}$ you feed $\mathcal{A}$ the algorithm will say NO. $\square$

Is your mind spinning yet?

✍

Exercise: *Prove that* IS *lies in the class* **NP**.

Here is another interpretation of **NP**. Imagine Luna wants to solve SAT but she has tried hard and failed. So she seeks help from an all powerful *prover* named Albus. More precisely, she seeks advice from Albus regarding an instance $\mathcal{I}$ of SAT to decide whether it is satisfiable or not. If $\mathcal{I}$ is indeed satisfiable, then Albus can find the solution (he is all powerful, remember) and write it down for Luna. Luna, however, is a bit paranoid and she is worried that Tom may be impersonating as Albus and trying to fool her. So she asks Albus to write down in a precise format and if there is any discrepancy, she just rejects.

So **NP** is the set of decision problems for which Luna can ask advice from Albus in a *precise format* such that for YES-instances, Albus can write down a solution which Luna can verify and convince herself that the instance is YES, while for NO-instances no solution from Albus, or from Tom impersonating as Albus, can ever fool Luna into believing that the instance is a YES-instance.

Remark: *"What's that N in NP?"* **NP** *does* **not** *stand for "not polynomial". Rather, it stands for "non-deterministic polynomial". If our algorithm (or Luna) is allowed to make divine guesses (advice from Albus), then for* YES*-instances there is a guess which leads to the correct resolution, while for* NO*-instance all guesses lead to rejection. The concept of* non-determinism *is a deep one and there are some computation models where non-determinism, in fact, doesn't give any power. Whether it does for polynomial time algorithms, is, quite literally,* **a million dollar question**.

✍

Exercise: *Prove* $\mathbf{P} \subseteq \mathbf{NP}$.

Question 1.

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

The famous **P** vs **NP** question asks whether $\mathbf{NP} \subseteq \mathbf{P}$ or not? Is there any problem in **NP** that is not in **P**? Is there any problem for which solutions can be verified efficiently but can't be found efficiently? We don't know.

4.1 NP-hardness and NP-completeness

Although we don't know whether $\mathbf{P} = \mathbf{NP}$ or not, we do know in a sense the *hardest* problems in $\mathbf{NP}$. This is one of the deepest facts in computer science and is due to Stephen Cook and independently, Leonid Levin.

Theorem 1 (Cook-Levin Theorem)**.** Let Π be any problem in $\mathbf{NP}$. Then, $\Pi \preceq_{\text{poly}}$ SAT.

We already saw SAT was in the class $\mathbf{NP}$. The above theorem says any problem in $\mathbf{NP}$ is "easier" than SAT. Thus, if we have a polynomial time algorithm for SAT, then there is a polynomial time algorithm for *all* problems in $\mathbf{NP}$. Amazing, isn't it? To prove $\mathbf{P} = \mathbf{NP}$, that is to argue about two sets, we just need to argue about one element in the set $\mathbf{NP}$. On the other hand, if we can prove SAT has no polynomial time algorithms, then we prove $\mathbf{P} \neq \mathbf{NP}$; we have found one element in $\mathbf{NP} \setminus \mathbf{P}$. Conjecture 1 asserts this latter state of affairs.

But we know there are harder problems than SAT. Lemma 3 and Lemma 4 shows this. And thus by transitivity Lemma 2 we get that for any problem $\Pi \in \mathbf{NP}$, we have $\Pi \preceq$ 3SAT and $\Pi \preceq$ IS as well. This motivates the following definitions.

Definition 5. A problem Π is $\mathbf{NP}$-hard if $\Pi' \preceq \Pi$ for any $\Pi' \in \mathbf{NP}$.

That is, Π is $\mathbf{NP}$-hard if its harder than all problems in $\mathbf{NP}$. Theorem 1 implies that (a) SAT is $\mathbf{NP}$-hard, and (b) to show Π is $\mathbf{NP}$-hard, all we need to show is SAT $\preceq \Pi$. Thus, 3SAT and IS are $\mathbf{NP}$-hard problems.

Definition 6. A problem Π is $\mathbf{NP}$-complete if it is (a) in $\mathbf{NP}$ and (b) is $\mathbf{NP}$-hard.

$\mathbf{NP}$-compelete problems form an *equivalence* class. By definition, for any two $\mathbf{NP}$-complete problems Π, Π', we have $\Pi \preceq \Pi'$ and $\Pi' \preceq \Pi$. Since the exercise above shows IS is in $\mathbf{NP}$ and we know it is $\mathbf{NP}$-hard, we get IS is indeed $\mathbf{NP}$-complete.

We end this lecture by saying that *all* problems don't lie in $\mathbf{NP}$. It may seem that verifying a solution should be simple. But that intuition is perhaps misplaced. Here is an example and you should convince that verifying the solutions don't seem simple.

#IS
Input: Same as IS – a graph G and a number K (note K could be as large as 2^n if n is the number of vertices in G).
Output: Decide whether or not the *number* of independent sets in G is $\leq K$.