CSE 347 Lecture 13

Reductions II: Relationships Between Problems

1. Polynomial-time reducibility
2. Vertex-cover is equivalent to independent set
3. Satisfiability

We introduced Max-Flow as an example of a problem that was not only interesting in its own right, but moreover "powerful" in that our algorithm for Max-Flow enables us to easily design algorithms for many other problems: we got algorithms for Maximum Bipartite Matching, Min-Cut, and, in turn, Image Segmentation. (And, the book gives many other examples.) Today, we'll formalize these kinds of relationships between problems, which will help us to understand better which problems can be solved by satisfactory algorithms, and which problems can't.

Polynomial-Time Reductions

The fundamental definition for these purposes captures what it means to solve one problem using an algorithm for another:

Definition A computational problem A *reduces to* a computational problem B ("$A \leq_p B$") if there is an algorithm for solving A that runs in polyno-

mial time when we are given an instruction (or "library call") that solves B in a single step of computation. (We must still pay for the steps to write the input to B and to read its output.)

So, for example, in last lecture, we showed Max-Bipartite-Matching $\leq_p$ Max-Flow and Image-Segmentation $\leq_p$ Min-Cut. The fact that both problems have polynomial-time algorithms was a consequence of our knowledge of the Ford-Fulkerson Algorithm, that enabled us to solve both Max-Flow and Min-Cut in polynomial time. In general:

Lemma Suppose A reduces to B (in polynomial time) and B has a polynomial-time algorithm. Then, A also has a polynomial-time algorithm.

Proof? Since A reduces to B, there is an algorithm for A that runs in some polynomial number of steps $p(n)$ on inputs of size n, given access to a subroutine for B. Suppose we replace these subroutine calls with our algorithm for B; we observe that this is immediately an algorithm for A. For the running time, we know that the algorithm for B runs in time $q(n)$ for some polynomial q. Now, since the algorithm for A has only $p(n)$ steps, it can't call B on an input of size larger than $p(n)$.

Therefore, the algorithm for B always runs in time $q(p(n))$ (where n is the size of the input to A). This is another polynomial — finally, we note that each step of our modified algorithm for A takes at most $q(p(n))$ times longer than the original, so it takes at most $p(n) \cdot q(p(n))$ steps total, which is indeed polynomial. ☒

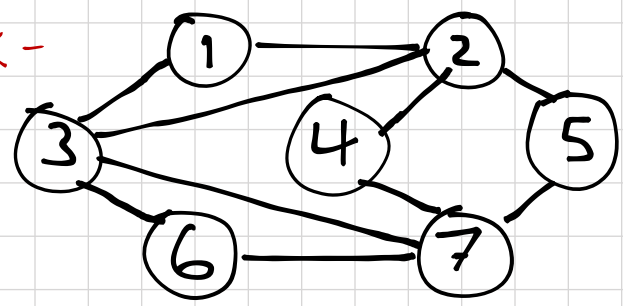
The construction is very natural, and indeed, this property that "polynomial time" does not change when we implement our instructions by further polynomial-time algorithms in a simpler instruction set is one of the fundamental reasons why we identify "efficient computation" with "polynomial time algorithms." But, it still raises the question of why we should be interested in such abstract notions of computation in the first place. The reason is that it enables us to meaningfully talk about problems that are <u>not</u> known to have "efficient algorithms," and sometimes determine that such algorithms (probably) can't be designed.

<u>Vertex-Cover is Equivalent to Independent Set</u>

For example, here are two problems for which we do <u>not</u> have efficient algorithms:

In a graph $G=(V,E)$, a set of nodes $S \subseteq V$ is <u>independent</u> if no two nodes in S are joined by an edge.

The independent set problem is: given a graph G and an integer k, does G have an independent set of size at least k? (i.e., we return a Boolean answer.) We note that we could have equally well have formulated the independent set problem as "Given a graph G, find the size of the largest independent set," since by using binary search on the size of the independent set, we can find the largest k for which the answer to the "decision" version is "yes," which then answers this second, "optimization" version. It turns out that we can further solve the "search" version — find some largest independent set — by iterating over vertices, connecting them to the rest of the graph, unless this diminishes the size of the largest independent set. So, the "decision" version we formulated captures the inherent difficulty of the problem. This simpler version will be convenient for us. So, for example, what is the largest independent set in this graph? {1,4,5,6}



Our second problem is the following. Given a graph $G=(V,E)$, a set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one end point in S. The vertex cover problem is: given a graph G and an integer k, is there a vertex cover of size at most k? Again, we could formulate "optimi-

zation" and "search" versions of this problem, but they could be solved by an algorithm for this "decision" problem. What is the smallest vertex cover in our example graph? {2,3,7}

On the surface, these two problems appear quite different. But, we will see that either both of them have efficient algorithms, or neither of them do.

Lemma In a graph $G=(V,E)$, $S \subseteq V$ is an independent set if and only if $V-S$ is a vertex cover.

Proof: ($\Rightarrow$) Let S be an independent set, and consider any edge $(u,v) \in E$. Since S is an independent set, either $u \notin S$ or $v \notin S$. Thus, at least one of u or v is in $V-S$, so $V-S$ is a vertex cover ◇($\Rightarrow$)

($\Leftarrow$) Suppose $V-S$ is a vertex cover, and let any two nodes u & v in S be given. Then, if $(u,v) \in E$, (u,v) would not have an endpoint in $V-S$, so $V-S$ would not be a vertex cover. Thus, we find that no pair is joined by an edge, so S is an independent set. ▨

Reductions between the problems are now immediate:

Theorem: Independent Set $\leq_p$ Vertex Cover and Vertex Cover $\leq_p$ Independent Set.

Proof? The algorithm to solve IndependentSet given VertexCover passes G and $n-k$ to the call to

Vertex Cover, and returns its answer. This is easy to do in time $O(n)$, and by our Lemma it is correct. Similarly, given a subroutine for Independent Set, we solve Vertex Cover by passing G and $n-k$ to the subroutine and returning the same answer. Again, this is $O(n)$ time and correct by the lemma ∎

Satisfiability

Here is another problem that is related to the previous two, but appears even more different. It originated in the study of automated reasoning.

A clause is an OR of terms (or "literals"), which are either a Boolean variable or a negated Boolean variable. A Conjunctive Normal Form formula ("CNF") is an AND of clauses. For example:
$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3)$ is a CNF. Since in this formula, each clause has (at most) two literals, we say that it is a 2-CNF. In general, if a CNF has only clauses of at most k literals, it is a k-CNF.

If we "plug in" Boolean values for the variables, then the formula either evaluates to true or false. For example, if we set all of the variables to true in our example, the second clause is false, and so the formula is false. But, if we set all of the variables to false, the formula is true. We say that any setting of the variables for which the formula

evaluates to true is a satisfying assignment. The satisfiability problem is: given a CNF φ, does φ have a satisfying assignment? We will be particularly interested in the special case of satisfiability in which φ is required to be a 3-CNF. We refer to this problem as 3-SAT. Next time, we will show $3\text{-SAT} \leq_p \text{Independent Set}$.