

## CME 305: Discrete Mathematics and Algorithms

Instructor: Professor Amin Saberi (saberi@stanford.edu)

February 5, 2009

# Lecture 9: Computation and Intractability

So far we've studied several problems for which 'efficient' algorithms are known, such as max-flow/min-cut, linear programming, bipartite matching, minimum spanning tree. By 'efficient', we mean solvable in polynomial time. However, it is often the case that if you change the problem statement for an 'easy' problem slightly, the problem becomes 'hard', i.e. that the best known algorithms take an exponential number of operations despite many years of study.

Some examples of "easy problems", and their "hard" counterparts:

"easy"	"hard"
shortest path	longest path
MST	min. spanning k-connected graphs, $k > 1$
2-coloring	k-coloring for $k > 2$

For many of the most fundamental problems in computer science, biology, combinatorics and elsewhere, there is no efficient algorithm known. Moreover, for many problems we don't know how to prove that no efficient algorithms exist. However, some progress has been made. A large class of these difficult problems have been shown to be "equivalent". This is the class of NP-completeness and the topic of lecture today.

## Polynomial Time Reducibility

Problem  $X$  is polynomially time reducible to problem  $Y$  ( $X \leq_p Y$ ) iff for every instance of  $X$  there is an algorithm that solves  $X$  with polynomially many operations and polynomially many calls to a "black box" that solves a given instance of  $Y$ . In other words, if for every instance of problem  $X$  we can represent it as a instance of  $Y$  (or sequence of such instances) using a polynomial number of computations, then we say that  $X$  is **harder** than  $Y$  and we write  $X \geq_p Y$ . If  $Y$  can be in turn solved through black box calls that solves instances of  $X$ , i.e.  $Y$  is also harder than  $X$ , then  $X$  and  $Y$  are **equivalent** and we write  $X =_p Y$ .

Instead of posing problems as *optimization* problems, we pose problems as yes/no or **decision** problems. An algorithm for a decision problem takes an input string  $s$  (a **certificate**) and returns either 1 or 0 depending on whether the input satisfies the constraints of the decision problem. Posing questions as decision problems provides a common framework and makes reductions between problems conceptually cleaner. Some examples of decision problems:

- Is the length of longest path  $\leq k$
- Is it possible to color this graph with 3 colors?

Generally, given a black box that solves a decision problem can also solve the related optimization problem. For example if we know the solution is  $\leq k$ , we can find the exact value through binary search.

**Example:**

A **vertex cover** in  $G(V, E)$  is a set of vertices  $S \subset V$  such that every edge in  $E$  has at least one endpoint in  $S$ . An **independent set** in  $G$  is a set of vertices  $S \subset V$  such that no two vertices in  $S$  share an edge.

**Claim 1** *A set  $S$  is a vertex cover iff its complement  $V - S$  is an independent set.*

**Corollary 1** *Max. indep. set = Min. vertex cover*  

$$p$$

## Problem Classes

For a given problem type, we can represent a particular instance of a decision problem as an input string  $s$ .  $s$  encodes all of the information about the instance of the problem we are interested in. Let  $X$  be the set of all problems represented by strings  $s$  that are solvable. Then the decision problem asks, is  $s \in X$ ? In other words, does there exist a solution to the problem instance represented by input string  $s$ ?

**P:** the class of decision problems that can be correctly decided in polynomial time. In other words, there exists some deterministic **verifier**  $A(s)$  that will return  $T$  iff  $s \in X$  in polynomial time.

**NP:** the class of all decision problems for which a YES certificate can be verified in polynomial time. More formally, a problem is in NP if for a string  $t$  (called a **certificate**) and there exists a polynomial-time deterministic verifier  $B(s, t)$  such that if  $s \in X$ , then there exists some  $t : |t| \leq |s|$  such that  $B(s, t)$  returns in  $T$ , otherwise (if  $s \notin X$ )  $B(s, t)$  returns  $F$  for all  $t$ .

**Co-NP:** the class of all decision problems for which a NO certificate can be verified in polynomial time.

We don't know whether  $P = NP$  or not. This is one of the "Millenium Prize Problems" by the Clay Institute, with a reward for solution of 1 million dollars.

**Theorem 1**  $P \subseteq NP$

**Proof:** This is straightforward from the definitions of P and NP. Given a polynomial verifier  $A(s)$ , we can construct a NP verifier  $B(s, t)$  simply by discarding  $t$  and running  $A(s)$ . If  $s \in X$ , then  $B(s, t)$  will return  $T$  for all certificates  $t$ , otherwise it will return false. Note that this *doesn't* say whether  $t$  is a “solution” of problem  $s$  (as respect to some arbitrary verifier  $B'(s, t)$ ), just that we can *construct* some a verifier  $B(s, t)$  that fits the requirements for NP. ■

A problem  $X$  is NP-complete iff

- $X \in \text{NP}$
- $\forall Y \in \text{NP } Y \leq X$

## Some Reductions

**Circuit Satisfiability** Given a circuit of logical operators (e.g. AND, NOT, OR), is there an assignment of T or F to variables that causes the circuit to output T?

**Theorem 2** *Circuit satisfiability is NP-complete.*

A circuit can be viewed as a tree  $T$ , where each leaf is a variable, and each internal node is a logical operation (AND, OR, NOT). The idea is that for any problem in NP, we can by definition take as input an assignment of variables and answer in a polynomial number of operations whether or not this assignment is valid or not. Cook's contribution was to show that for any problem, this verification process can be encoded as a polynomial-sized circuit. In other words, if  $X \in \text{NP}$ , then we can design a polynomial-sized circuit  $Y$  such that for a given variable assignment  $x$ ,  $Y$  outputs true if and only if  $x$  is a truth assignment to  $X$ .

The circuit satisfiability problem then asks, is there an assignment to the variables so that the answer is “TRUE”? Clearly, if we can solve this problem, we know whether or not there exists a truth assignment to the original problem  $X$ .

Finding the first NP-Complete problem made it much easier to establish NP-completeness of other problems. In order to prove that  $Y \in \text{NP}$  NP-Complete, it is sufficient to show that  $\text{circuit-SAT} \leq_p Y$ , i.e. that we can 'reduce'  $Y$  to a circuit satisfiability problem.

A circuit that is a series of OR clauses joined by AND clauses is said to be in **conjunctive normal form** (CNF). For example:

$$(x_1 \wedge \bar{x}_2 \wedge \bar{x}_3) \vee (\bar{x}_2 \wedge x_1) \vee (x_4 \wedge x_5)$$

**Satisfiability:** Given a boolean formula in conjunctive normal form (CNF), is there an assignment of T or F such that the formula is satisfied?

It is easy to see that

- $\text{SAT} \in \text{NP}$  (we can verify a solution in poly. time).
- $\text{SAT} \stackrel{p}{\geq} \text{circuit-SAT}$  (we can solve circuit-SAT using SAT).

Therefore, SAT is NP-C.

**3-SAT:** A SAT problem in which every clause has exactly 3 literals.

**Claim 2** *Every SAT formula of size  $n$  can be described as a 3-SAT formula of size polynomial in  $n$ .*

**Proof:** We can split any clause  $C$  with  $k > 3$  into two clauses, the first with 2 variables from  $C$ , the second with  $k - 2$  variables from  $C$ . We introduce a dummy variable  $d_c$  into the first new clause, and its complement  $\bar{d}_c$  into the second clause. It is easy to see that an assignment that satisfies the two new clauses will give us a solution to the original clause. Repeat this until all clauses are of size 3. ■

**Corollary 2**  $3\text{-SAT} \in \text{NP-C}$ .

**Theorem 3** *Maximum independent set  $\stackrel{p}{\leq}$  3-SAT*

**Proof:** For each clause  $C_i$ , construct a 3-cycle with vertices  $v_{i1}, v_{i2}, v_{i3}$  representing the 3 variables in clause  $i$ , say  $x_j, x_k, x_l$ . For each node representing variable  $x_j$ , put an edge between it and each vertex representing its complement  $\bar{x}_j$ . Call this constructed graph  $G$ .

Given a satisfying assignment to  $C = \{C_1, \dots, C_k\}$ , we can construct an independent set in  $G$  by picking one true variable in each  $C_j$ . Also, given an independent set in  $G$ , we can reconstruct an assignment in  $C$  by setting the variables corresponding to the independent set vertices to be  $T$  and assigning the rest arbitrarily. Therefore, a maximum satisfying assignment to  $C$  corresponds directly to a maximum independent set in  $G$ , so  $C$  has a satisfying assignment if and only if  $G$  has an independent set. ■

## Approximation Algorithms

What do we do if the problem we wish to solve is intractable? There are 3 basic approaches:

- Exploit special problem structure: perhaps we don't need to solve the *general* case of the problem but rather a tractable special version

- Heuristics: procedures that hopefully give reasonable estimates but for which there are no proven results
- Approximation algorithms: procedures for which are proven to give solutions within a factor of the optimum

We will focus in this lecture on the last approach.

**Definition:** An algorithm is a factor  $\alpha$  approximation for a problem iff for every instance of the problem it can find a solution within factor  $\alpha$  of the optimum solution.

One of the most difficult challenges in the analysis of approximation algorithms is to find a *lower bound* of value of the optimal solution.

**Example: (Minimum Vertex Cover)** Given a graph  $G(V, E)$ , find a subset  $S \subseteq V$  with minimum cardinality such that every edge in  $E$  has at least one endpoint in  $S$ .

---

**Algorithm 1** 2-Approximation for Vertex Cover

---

Find a maximal matching  $M$  in  $G$ .

Output the endpoints of edges in  $M$ .

---

**Claim 3** *The solution of the previous algorithm is feasible.*

**Proof:** Suppose not. This means that there is some edge  $e$  that is not covered, does not share endpoints with any edge in  $M$ . This means that  $M \cup e$  is a larger matching, meaning that  $M$  was not maximal. This is a contradiction. ■

**Claim 4** *The cardinality of the solution of the algorithm is at most twice the optimum for all maximal matchings  $M$ .*

**Proof:** We need to take at least one vertex from every edge in  $M$ , since the edges in  $M$  are independent. This means that  $|M| \leq OPT$ . By the previous claim,  $2|M|$  is a feasible solution, meaning that

$$|M| \leq OPT \leq 2|M|$$

■