MS&E-212 Combinatorial Optimization

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

Lecture 9: Intractability

So far we have been studying several problems for which "efficient" or polynomial-time algorithms are known. For example, we studied max-flow/min-cut, linear programming, bipartite matching, or minimum spanning tree. We also observed that they can be used as building blocks for designing algorithms for a variety of applications.

Nevertheless, there are several problems for which there are no polynomial-time algorithms are known. In fact, it is often the case that if you slightly change the problem statement for an "easy" problem, the problem becomes "hard", i.e. the best known algorithms take an exponential number of operations for solving this problem in the worst case.

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

"easy"	"hard"
shortest path	longest path
MST	min. Steiner tree; min. spanning k-connected graphs, $k > 1$
2-coloring	k-coloring for $k > 2$
Matching in bipartite graphs	3D matching

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 ($X \leq_p Y$) if and only if 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 an instance of Y (or a sequence of such instances) using a polynomial number of computations, then we say that Y is **as hard as** X and we write $X \leq_p Y$. If Y can also be solved through black box calls that solve instances of X, i.e. Y is also as hard as 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, 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 $S \subseteq V$ such that every edge in E has at least one endpoint in S. An **independent set** in G is a set of vertices $S \subseteq V$ such that no two vertices in S share an edge.

Claim 1 *A set S is a vertex cover if and only if its complement $V \setminus S$ is an independent set.*

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

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 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, just that we can *construct* some a verifier $B(s, t)$ that fits the requirements for NP. ■

NP-Complete Problems: A problem X is NP-Complete or $X \in$ NP-C if and only if it is in NP and it is as hard as *any problem* in NP.

- $X \in$ NP
- $\forall Y \in$ NP $Y \leq_p X$

A consequence of this definition is that if we had a polynomial time algorithm for X, we could solve all problems in NP in polynomial time. It is worth mentioning that a problem satisfying the second condition is said to be NP-hard, whether or not it satisfies the first condition.

The concept of NP-completeness was introduced in 1971 by Stephen Cook. This is perhaps the most well-known complexity class outside of computer science. In the celebrated Cook-Levin theorem (independently proved by Leonid Levin), Cook proved that the satisfiability problem is NP-complete. Shortly after, Richard Karp proved that several other problems were also NP-complete; thus there is a class of NP-complete problems (besides the Boolean satisfiability problem). Since Cook's original results, thousands of other

problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete. We refer the interested reader to Garey and Johnson's Computers and Intractability: A Guide to the Theory of NP-Completeness, or "A compendium of NP optimization problems" which is available online. In the rest of this note, we will cover a few basic definitions and reductions and leave the details to a computational complexity course.

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

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

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

- SAT $\in$ NP (we can verify a solution in poly. time).
- SAT $\leq_p$ 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-SAT $\in$ NP-C.*

Theorem 3 *Maximum independent set $\geq_p$ 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 $C = \{C_1, \ldots, C_k\}$, 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 maxmimum 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. ■