COMS W3261
Computer Science Theory
Chapter 10 Notes

Alexander Roth

2014–11–08

Intractable Problems

Recall

- The problems solvable in polynomial time on a typical computer are exactly the same as the problems solvable in polynomial time on a Turing machine.

The Classes $\mathcal{P}$ and $\mathcal{NP}$

Problems Solvable in Polynomial Time

A Turing machine M is said to be of *time complexity* $T(n)$ if whenever M is given an input w of length n, M halts after making at most $T(n)$ moves, regardless of whether or not M accepts. A language L is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM M of time complexity $T(n)$.

An Example: Kruskal's Algorithm

There is a well-known "greedy" algorithm, called *Kruskal's Algorithm*, for finding a minimum-weight spanning tree.

1. Maintain for each node the *connected component* in which the node appears, using whatever edges of the tree have been selected so far. Initially, no edges are selected, so every node is then in a connected component by itself.

2. Consider the lowest-weight edge that has not yet been considered; break ties any way you like. If this edge connects two nodes that are currently in different connected components then:

(a) Select that edge for the spanning tree, and

(b) Merge the two connected components involved, by changing the component number of all nodes in one of the two components to be the same as the component number of the other.

If, on the other hand, the selected edge connects two nodes of the same component, then this edge does not belong in the spanning tree; it would create a cycle.

3. Continue considering edges until either all edges have been considered, or the number of edges selected for the spanning tree is one less than the number of nodes. Note that in the latter case, all nodes must be in one connected component, and we can stop considering edges.

In the theory of intractability, we generally want to argue that a problem is hard, not easy, and the fact that a yes-no version of a problem is hard implies that a more standard version, where a full answer must be computed, is also hard.

Problem elements must be encoded suitably to work with a Turing machine. The effect of this requirement is that inputs to Turing machines are generally slightly longer than the intuitive "size" of the input. However, there are two reasons why the difference is not significant:

1. The difference between the size as a TM input string and as an informal problem input is never more than a small factor, usually the logarithm of the input size.

2. The length of a string representing the input is actually a more accurate measure of the number of bytes a real computer has to read to get its input.

Nondeterministic Polynomial Time

A language L is in the class $\mathcal{NP}$ if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = L(M)$, and when M is given an input of length n, there are no sequences of more than $T(n)$ moves of M.

An $\mathcal{NP}$ Example: The Traveling Salesman Problem

Consider the *Traveling Salesman Problem (TSP)*. The input to TSP is the same as to MWST, a graph with integer weights on the edges, and a weight limit W. The question asked is whether the graph has a "Hamilton circuit" of total weight at most W. A *Hamilton circuit* is a set of edges that connect the nodes into a single cycle, with each node appearing exactly once.

Polynomial-Time Reductions

In the theory of intractability, we shall use *polynomial-time reductions* only. A reduction from P_1 to P_2 is polynomial-time if it takes time that is some polynomial in the length of the P_1 instance. The P_2 instance will be of a length that is polynomial in the length of the P_1 instance.

NP-Complete Problems

Let L be a language. We say L is *NP-complete* if the following statements are true about L:

1. L is in $\mathcal{NP}$.
2. For every language L' in $\mathcal{NP}$ there is a polynomial-time reduction of L' to L.

Theorem 1. If P_1 is NP-complete, P_2 is in $\mathcal{NP}$, and there is a polynomial-time reduction of P_1 to P_2, then P_2 is NP-complete.

NP-Hard Problems

Some problems L are so hard that although we can prove condition (2) of the definition of NP-completeness (every language in $\mathcal{NP}$ reduces to L in polynomial time), we cannot prove condition (1): that L is in $\mathcal{NP}$. If so, we call L *NP-hard*. It is generally acceptable to use "intractable" to mean "NP-hard."

An NP-Complete Problem

The Satisfiability Problem

The *boolean expressions* are built from:

1. Variables whose values are boolean; i.e., they either have the value 1 (true) or 0 (false).
2. Binary operators $\wedge$ and $\vee$, standing for the logical AND and OR of two expressions.
3. Unary operator $\neg$ standing for logical negation.
4. Parentheses to group operators and operands, if necessary to alter the default precedence of operators: $\neg$ highest, then $\wedge$, and finally $\vee$.

A *truth assignment* for a given boolean expression E assigns either true or false to each of the variables mentioned in E. The *value* of expression E given a truth assignment T, denoted $E(T)$, is the result of evaluting E with each variable x replaced by the value $T(x)$ that T assigns to x.

A truth assignment T *satisfies* boolean expression E if $E(T) = 1$; i.e., the truth assignment T makes expression E true. A boolean expression E is said to be *satisfiable* if there exists ast least one truth assignment T that satisfies E.

The *satisfiability problem* is:

- Given a boolean expression, is it satisfiable?

Representing SAT Instances

Since there are an infinite number of symbols that could in principle appear in a boolean expression, we have a familiar problem of having to devise a code with a fixed, finite alphabet to represent expressions with arbitrarily large number of variables. Only then can we talk about SAT as a "problem," that is, as a language over a fixed alphabet consisting of codes for those boolean expressions that are satisfiable. The code wee shall use is as follows:

1. The symbols $\wedge$, $\vee$, $\neg$, (, and) are represented by themselves.
2. The variable x_i is represented by the symbol x followed by 0's and 1's that represent i in binary.

Thus, the alphabet for the SAT problem/language has only eight symbols. All instance of SAT are strings in this fixed, finite alphabet.

Notice that the length of a coded boolean expression is approximately the same as the number of positions in the expression, counting each variable occurrence as 1.

NP-Completeness of the SAT Problem

To prove a problem is NP-complete, we need first to show that it is in $\mathcal{NP}$. Then, we must show that every language in $\mathcal{NP}$ reduces to the problem in question In general, we show the second part by offering a polynomial-time reduction from some other NP-complete problem, and then invoking Cook's Theorem.

Theorem 2. (Cook's Theorem) SAT is NP-complete.

A Restricted Satisfiability Problem

Normal Forms for Boolean Expressions

The following are three essential definitions:

- A *literal* is either a variable, or a negated variable.
- A *clause* is the logical OR of one or more literals.
- A boolean expression is said to be in *conjunctive normal form* or *CNF* if it is the AND of clauses.

An expression is said to be in *k-conjunctive normal form* (k-CNF) if it is the product of clauses, each of which is the sum of exactly k distinct literals. For instance, $(x + \overline{y})(y + \overline{z})(z + \overline{x})$ is in 2-CNF, because each of its clauses has exactly two literals.

- CSAT is the problem: given a boolean expression in CNF, is it satisfiable?
- kSAT is the problem: given a boolean expression in k-CNF, is it satisfiable?

Converting Expressions to CNF

Two boolean expressions are said to be *equivalent* if they have the same result on any truth assignment to their variables. If two expressions are equivalent, then surely either both are satisfiable

Our first step is to push $\neg$'s below $\wedge$'s and $\vee$'s. The rules we need are:

1. $\neg(E \wedge F) \rightarrow \neg(E) \vee \neg(F)$. This rule, one of *DeMorgan's laws*, allows us to push $\neg$ below $\wedge$. Note that as a side-effect, the $\wedge$ is changed to an $\vee$.
2. $\neg(E \vee F) \rightarrow \neg(E) \wedge \neg(F)$. The other "DeMorgan's law" pushes $\neg$ below $\vee$. The $\vee$ is changed to $\wedge$ as a side- effect.
3. $\neg(\neg(E)) \rightarrow E$. This *law of double negation* cancels a pair of $\neg$'s that apply to the same expression.

Theorem 3. Every boolean expression E is equivalent to an expression F in which the only negations occur in literals; i.e., they apply directly to variables. Moreover, the length of F is linear in the number of symbols of E and F can be constructed from E in polynomial time.

NP-Completeness of CSAT

Now, we need to take an expression E that is the AND and OR of literals and convert it to CNF.

Theorem 4. CSAT is NP-complete.

NP-Completeness of 3SAT

Theorem 5. 3SAT is NP-complete

Summary of Chapter 10

The Classes $\mathcal{P}$ and $\mathcal{NP}$ $\mathcal{P}$ consists of all those languages or problems accepted by some Turing machine that runs in some polynomial amount of time, as a function of its input length. $\mathcal{NP}$ is the class of languages or problems that are accepted by nondeterministic TM's with a polynomial bound on the time taken along any sequence of nondeterministic choices.

The $\mathcal{P} = \mathcal{NP}$ Question It is unknown whether or not $\mathcal{P}$ and $\mathcal{NP}$ are really the same classes of languages, although we suspect strongly that there are languages in $\mathcal{NP}$ that are not in $\mathcal{P}$.

Polynomial-Time Reductions If we can transform instance of one problem in polynomial time into instances of a second problem that has the same answer – yes or no – then we say the first problem is polynomial-time reducible to the second.

NP-Complete Problems A language is NP-complete if it is in $\mathcal{NP}$, and there is a polynomial-time reduction from each language in $\mathcal{NP}$ to the language in question. We believe strongly that none of the NP-complete problems are in $\mathcal{P}$, and the fact that no one has ever found a polynomial-time algorithm for any of the thousands of known NP-complete problems is mutually re-enforcing evidence that none are in $\mathcal{P}$.

NP-Complete Satisfiability Problems Cook's theorem showed the first NP-complete problem – whether a boolean expression is satisfiable – by reducing all problems in $\mathcal{NP}$ to the SAT problem in polynomial time. In addition, the problem remains NP-complete even if the expression is restricted to consist of a product of clauses, each of which consists of only three literals – the problem 3SAT.

Other NP-Complete Problems There is a vast collection of known NP-complete problems; each is proved NP-complete by a polynomial-time reduction from some previously known NP-complete problem. We have given reductions that show the following problems NP-complete: the independent set, node cover, directed and undirected versions of the Hamilton circuit problem, and the traveling-salesman problem.