Chapter 11

Introduction to Computational Complexity

Introduction to Computational Complexity

- A decision problem is decidable if there is an algorithm that can answer it in principle
- In this chapter, we try to identify the problems for which there are *practical* algorithms
  - Ones that can answer reasonable-size instances in a reasonable amount of time
- The *satisfiability problem* is decidable, but the known algorithms aren't much of an improvement on the brute-force algorithm that takes exponential time

The Time Complexity of a Turing Machine, and the Set P

- The set P is the set of problems that can be decided by a TM in *polynomial time*, as a function of the instance size. (Brute-force algorithms tend to be exponential)
- NP is defined similarly, except that we allow the use of a *nondeterministic* TM
- Most people assume that NP is a larger set, but no one has been able to demonstrate that $P \neq NP$
- We discuss NP-complete problems, which are hardest problems in NP, and show that the satisfiability problem is one of these

The Time Complexity of a Turing Machine, and the Set P (cont'd.)

- A TM deciding a language $L \subseteq \Sigma^*$ solves a decision problem: Given $x \in \Sigma^*$, is $x \in L$?
  - A measure of the size of the problem is the length of the input string x

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- Definition 11.1: Suppose T is a TM with input alphabet Σ that eventually halts on every input string
  - The *time complexity* of T is the function $\tau_T : \mathbb{N} \to \mathbb{N}$, where $\tau_T(n)$ is defined by considering, for every input string of length n in Σ^*, the number of moves T makes on that string before halting, and letting $\tau_T(n)$ be the maximum of these numbers
  - When we refer to a TM with a certain time complexity, it will be understood that it halts on every input

The Time Complexity of a Turing Machine, and the Set P (cont'd.)

- Definition 11.4: If f and g are partial functions from $\mathbb{N}$ to $\mathbb{R}^+$; that is, both functions have values that are nonnegative real numbers wherever they are defined
  - We say that $f = O(g)$, or $f(n) = O(g(n))$ (which we read "f is big-oh of g" or "$f(n)$ is big-oh of $g(n)$") if, for some positive numbers C and N, $f(n) \leq C\,g(n)$ for every $n \geq N$
  - For example, every polynomial of degree k with positive leading coefficient is $O(n^k)$

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- An instance of the *satisfiability problem* is a Boolean expression
  - It involves Boolean variables $x_1, x_2, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$
  - It is in conjunctive normal form (the conjunction of several clauses, each of which is a disjunction)
- Is there an assignment of truth values to the variables that satisfies the expression (makes it true)?
  - This problem is clearly decidable
    - We could simply try every possible assignment of values to variables

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- The *traveling salesman problem* considers *n* cities that a salesman must visit, with a distance specified for every pair of cities
  - It's simplest to formulate this as an optimization problem
    - Determine the order that minimizes the total distance traveled
  - We can turn this into a decision problem by introducing a variable *k* and asking whether there is an order in which the cities could all be visited by traveling no more than distance *k*

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- There's a brute-force solution to this problem too
  - Consider all $n!$ possible permutations of the cities
- With current hardware we can solve very large problems, if the problems require time $O(n)$
- We can still solve largish problems if they take time $O(n^2)$ or even $O(n^3)$
- Exponential problems are another story
  - If the problem really requires time proportional to 2^n, then doubling the speed of the machine only allows us to increase the size of the problem by 1!

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- Showing that a brute-force approach takes a long time does not necessarily mean that the problem is complex
  - The satisfiability problem and the traveling salesman problem are assumed to be hard, not because the brute-force approach takes exponential time, but because no one has found a way of solving either problem that *doesn't* take at least exponential time

The Time Complexity of a Turing Machine, and the Set *P* (cont'd.)

- What constitutes a *tractable* problem?
  - The most common answer is those that can be solved in polynomial time on a TM or other computer
  - One reason for this characterization is that it is relatively robust, as problems that can be solved in polynomial time on any computer can be solved in polynomial time on a TM as well, and vice-versa

The Time Complexity of a Turing Machine, and the Set P (cont'd.)

- Definition 11.5: P is the set of languages L such that for some TM T deciding L and some $k \in \mathbb{N}$, $\tau_T(n) = O(n^k)$
- The satisfiability and traveling salesman problems seem to be good candidates for real-life problems that are not in P

The Set *NP* and Polynomial Verifiability

- The satisfiability problem seems like a hard problem
  - Testing a potential answer is easy, but there are an exponential number of potential answers
- We can approach this problem nondeterministically
  - We guess an answer (a particular truth assignment) and then test it deterministically
  - This can be done in polynomial time

The Set *NP* and Polynomial Verifiability (cont'd.)

- Definition 11.6: If T is an NTM with input alphabet Σ such that, for every $x \in \Sigma^*$, every possible sequence of moves of T on input x eventually halts, the time complexity $\tau_T : \mathbb{N} \to \mathbb{N}$ is defined as follows:
  - Let $\tau_T(n)$ be the maximum number of moves T can possibly make on any input string of length n before halting
  - As before, if we speak of an NTM as having a time complexity, we are assuming implicitly that no input string can cause it to loop forever

The Set *NP* and Polynomial Verifiability (cont'd.)

- Definition 11.7: *NP* is the set of languages L such that for some NTM T that cannot loop forever on any input, and some integer k, T accepts L and

  $\tau_T(n) = O(n^k)$

  - We say that a language in *NP* can be accepted in *nondeterministic polynomial time*
  - It is clear that $P \subseteq NP$
  - The *Sat* problem is in *NP* (the "guess-and-test" strategy is typical of problems in *NP*, and we can formalize this by constructing an appropriate NTM)

The Set *NP* and Polynomial Verifiability (cont'd.)

- Definition 11.10: If $L \subseteq \Sigma^*$, we say that a TM T is a *verifier* for L if:
  - T accepts a language $L_1 \subseteq \Sigma^*\{\$\}\Sigma^*$, T halts on every input, and
  - $L = \{x \in \Sigma^* \mid \text{for some } a \in \Sigma^*, x\$a \in L_1\}$ (we will call such a value a a *certificate* for x)
- A verifier T is a *polynomial-time verifier* if:
  - There is a polynomial p such that for every x and every a in Σ^*, the number of moves T makes on the input string $x\$a$ is no more than $p(|x|)$

The Set NP and Polynomial Verifiability (cont'd.)

- Theorem 11.11: For every language $L \in \Sigma^*$, $L \in NP$ if and only if L is polynomially verifiable
  - i.e., there is a polynomial-time verifier for L
- Proof: See book
- A verifier for the satisfiability problem could take a specific truth assignment as a certificate; the traveling salesman problem could take a permutation of the cities as a certificate

Polynomial-Time Reductions and *NP*-Completeness

- Just as we can show that a problem is decidable by reducing it to another one that is, we can show that a language is in P by reducing it to another that is
  - In the case of decidability, we only needed the reduction to be computable
  - Here we need the reduction function to be computable in polynomial time

Polynomial-Time Reductions and *NP*-Completeness (cont'd.)

- Definition 11.12: If L_1 and L_2 are languages over respective alphabets Σ_1 and Σ_2, a *polynomial-time reduction* from L_1 to L_2 is a function $f : \Sigma_1^* \to \Sigma_2^*$ satisfying two conditions
  - First: for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$
  - Second: f can be computed in polynomial time
    - i.e., there is a TM with polynomial time complexity that computes f
- If there is a polynomial-time reduction from L_1 to L_2, we write $L_1 \leq_p L_2$ and say that L_1 is polynomial-time reducible to L_2.

Polynomial -Time Reductions and *NP*-Completeness (cont'd.)

- Theorem 11.13:
  - Polynomial-time reducibility is transitive:
    - If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$
  - If $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$
- Proof sketch:
  - For the first statement, simply use the composition of the reduction functions
  - For the second statement, simply combine the TM that accepts L_2 and the one that computes the reduction f

Polynomial-Time Reductions and *NP*-Completeness (cont'd.)

- Definition 11.16: A language L is *NP-hard* if $L_1 \leq_p L$ for every $L_1 \in NP$; L is *NP-complete* if $L \in NP$ and L is *NP*-hard
- Theorem 11.17:
  - If L and L_1 are languages such that L is *NP*-hard and $L \leq_p L_1$, then L_1 is also *NP*-hard
  - If L is any *NP*-complete language, then $L \in P$ if and only if $P = NP$
- Proof of Theorem 11.17: both parts follow from Theorem 11.13

The Cook-Levin Theorem

- Theorem 11.18:
  - The language *Satisfiable* (or the corresponding decision problem *Sat*) is *NP*-complete
- Proof:
  - We know that *Satisfiable* is in *NP*, so we need to show that every language $L \in NP$ is reducible to *Sat*
  - We do this by using a TM T that accepts L; the reduction considers the details of T and takes a string x to a Boolean formula that is satisfiable if and only if x is accepted by T
  - The details are complex and can be found in the book

Some Other *NP*-Complete Problems

- Theorem 11.19:
  - The complete subgraph problem (Given a graph G and an integer k, does G have a complete subgraph with k vertices?) is *NP*-complete.
- Proof sketch:
  - By reduction from *Satisfiability*. For a Boolean expression x in conjunctive normal form, a graph can be constructed with vertices corresponding to occurrences of literals in x, and edges and an integer k chosen so that x is satisfiable if and only if the graph has a complete subgraph with k vertices

Some Other *NP*-Complete Problems

- The problem *3-Sat* is the same as *Sat* except that every conjunct in the CNF expression is assumed to be the disjunction of three or fewer literals
- Theorem 11.20: *3-Sat* is *NP*-complete. Proof sketch:
  - *3-Sat* is in *NP* because Sat is
  - To get a reduction f from *Sat* to *3-Sat*, we let $f(x)$ involve the variables in x as well as new ones
  - The trick is to incorporate the new variables so that
    - For every satisfying truth assignment to the variables of x, some assignment to the new variables makes $f(x)$ true
    - For every nonsatisfying assignment to the variables of x, no assignment to the new variables makes $f(x)$ true

Some Other *NP*-Complete Problems (cont'd.)

- A *vertex cover* for a graph G is a set C of vertices such that every edge of G has an endpoint in C
- The *vertex cover problem* is this: Given a graph G and an integer k, is there a vertex cover for G with k vertices?
- A *k-coloring* of G is an assignment to each vertex of one of the k colors so that no two adjacent vertices are colored the same
- The *k-colorability problem*: Given G and k, is there a k-coloring of G?

Some Other *NP*-Complete Problems (cont'd.)

- Theorem 11.21: The vertex cover problem is *NP*-complete
- Proof: We show that the problem is *NP*-hard by reducing the complete subgraph problem to it
  - The problem is clearly in *NP*
- Theorem 11.22: The *k-colorability* problem is *NP*-complete
- Proof: by reducing *3-Sat* to *k-colorability*
  - This problem is also clearly in *NP*

Some Other *NP*-Complete Problems (cont'd.)

- We now have five problems that are *NP*-complete
- There are thousands of others that are also known to be *NP*-complete
- Many real-life decision problems require some kind of solution
  - If a polynomial-time algorithm does not present itself, it is worth checking whether the problem is *NP*-complete
  - If so, finding such an algorithm will be as hard as proving that $P = NP$