CS 4410

Automata, Computability, and Formal Language

Chapter 14: An Overview of Computational Complexity

Dr. Xuejun Liang

Chapter 14
An Overview of Computational Complexity

1. Efficiency of Computation
2. Turing Machine Models and Complexity
3. Language Families and Complexity Classes
4. The Complexity Classes P and NP
5. Some NP Problems
6. Polynomial-Time Reduction
7. NP-Completeness and an Open Question

Learning Objectives
At the conclusion of the chapter, the student will be able to:

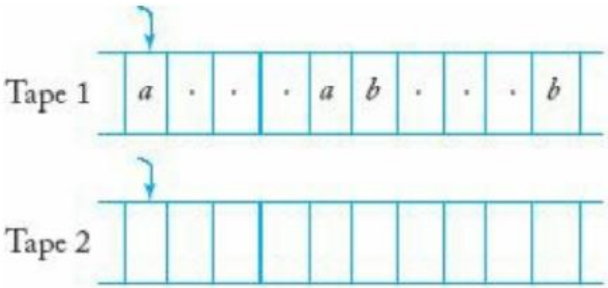
- Explain the concept of computational complexity as it relates to Turing machines
- Describe deterministic and nondeterministic solutions to the SAT problem
- Determine if a Boolean expression in CNF is satisfiable
- Describe the efficiency of standard Turing machines that simulate two-tape machines and of those that simulate nondeterministic machines
- Define the complexity classes P and NP, as well as the relationship between P and NP
- Explain the concepts of intractability and NP-completeness
- List some well-known NP-complete problems
- Discuss the significance and status of the P = NP? question

Efficiency of Computation

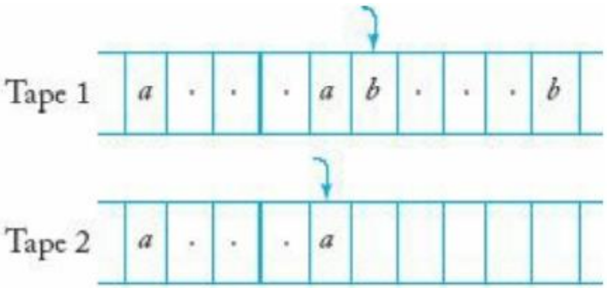
- *Computational complexity* is the study of the efficiency of algorithms
- When studying the time requirements of an algorithm, the following assumptions are made:
  - The algorithm will be modeled by a Turing machine
  - The size of the problem will be denoted by n
  - When analyzing an algorithm, the focus is on its general behavior, particularly as the size of the problem increases
- A computation has time-complexity *T(n)* if it can be completed in no more than *T(n)* moves on some Turing machine

Turing Machine Models and Complexity

- Although different models of Turing machines are equivalent, the efficiency of a computation can be affected by the number of tapes available and by whether it is deterministic or nondeterministic
- Example 12.7: Consider language $L = \{a^n b^n : n \geq 1\}$
  - Single-tape Turing machine (recall example 9.7). For $w = a^n b^n$, it takes roughly 2n steps to match each a with the corresponding b. Therefore, the whole computation takes $O(n^2)$ moves.
  - Using a two-tape Turing machine as below, the whole computation takes O(n) moves.



(a) Initial tapes



(b) Tapes after copying of a's

Satisfiability Problem (SAT)

- Boolean expression in **conjunctive normal form** (CNF).

$$e = t_i \wedge t_j \wedge \ldots \wedge t_k.$$ $$s_q = x_i \text{ or } s_q = \overline{x_i}$$

$$t_i = s_l \vee s_m \vee \ldots \vee s_p$$ variables $x_1, x_2, \ldots, x_n,$

- The *Satisfiability Problem* (SAT):

  Given a Boolean expression e in conjunctive normal form, find an assignment of values to the variables so that e is true

- For example, the expression $e_1 = (\overline{x}_1 \vee x_2) \wedge (x_1 \vee x_3)$ is true when $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$
- However, the expression $e_2 = (x_1 \vee x_2) \wedge \overline{x}_1 \wedge \overline{x}_2$ is not satisfiable

Solving the Satisfiability Problem

- A deterministic algorithm would take all possible values for the n variables and evaluate the expression for each combination
- Since there are 2^n possibilities, the deterministic solution has exponential time complexity
- A nondeterministic algorithm would guess the value of each of the n variables at each step and evaluate each of the 2^n possibilities simultaneously, thus resulting in an O(n) algorithm
- There is no known nonexponential deterministic algorithm for solving the SAT problem

Simulation of a Two-Tape Machine

- Theorem 14.1 states that, if a two-tape machine can carry out a computation in n steps, the computation can be simulated by a standard Turing machine in $O(n^2)$ moves
- To simulate the two-tape computation, the standard machine would
  - Keep a description of the two-tape machine on its tape
  - For each two-tape move, search the entire active area of its tape
- After n moves, the active area has a length of at most O(n), so the entire simulation takes $O(n^2)$ moves

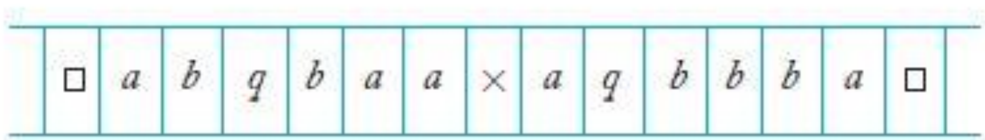


FIGURE 14.1

Simulation of a Nondeterministic Machine

- Theorem 14.2 states that, if a nondeterministic machine can carry out a computation in n steps, the computation can be carried out by a standard Turing machine in $O(k^{an})$ moves, where k and a are independent of n
- To simulate the nondeterministic computation, the standard machine would keep track of all possible configurations, searching and updating the entire active area of its tape
- If k is the maximum branching factor for the nondeterministic machine, after n steps there are at most k^n possible configurations, and the length of each configuration is O(n)
- Therefore, to simulate one move, the standard machine must search an active area of length $O(nk^n)$

Complexity Classes of Languages

- Definition 14.1
  - We say that a Turing machine M accepts a language L in time T(n) if every w in L with |w| = n is accepted in T(n) moves.
- Definition 14.2
  - A language L is said to be a member of the class DTIME (T(n)) if there exists a deterministic multitape Turing machine that accepts L in time O(T(n)).
  - A language L is said to be a member of the class NTIME (T(n)) if there exists a nondeterministic multitape Turing machine that accepts L in time O(T(n)).
- Theorem 14.3
  - For every integer $k \geq 1$, DTIME (n^k) $\subset$ DTIME (n^{k+1}).

Considerations about Language Complexity Classes

- There exists an infinite number of properly nested complexity classes DTIME (n^k), $k = 1, 2, \ldots$ These complexity classes have little connection to the familiar Chomsky hierarchy and it seems difficult to get any insight into the nature of these classes.
- The particular model of Turing machine used affects the complexity of the associated algorithms. It is difficult to determine which variation to use as the best model of an actual computer.
- The efficiency differences between deterministic and nondeterministic algorithms can be much more significant than differences between alternative deterministic algorithms involving different numbers of available tapes

The Complexity Classes P and NP

- There are two famous complexity classes associated with languages: *P* and *NP*
- *P* is the *set of all languages that are accepted by some deterministic Turing machine in polynomial time*, without any regard to the degree of the polynomial.
- *NP* is the *set of all languages that are accepted by some nondeterministic Turing machine in polynomial time.*

We say that a Turing machine M accepts a language L in time T(n) if every w in L with |w| = n is accepted in T(n) moves

The Relationship Between P and NP

- Obviously, $P \subseteq NP$
- What is not known is whether P is a proper subset of NP, in other words,

  is $P \subset NP$ or $P = NP$?
- While it is generally believed that there are languages in NP which are not in P, no one has yet found a conclusive example
- Because of its significance on the feasibility of certain computations, this question remains *the most fundamental unresolved problem in theoretical computer science*

Intractability

- A problem is *intractable* if it has such high resource requirements that practical solutions are unrealistic, although the problem may be computable in principle
- Algorithms for solving intractable problems consume an extraordinary amount of time for nontrivial values of n on any computer available now or in the foreseeable future
- According to the *Cook-Karp thesis*, a problem in P is tractable, and one not in P is intractable

Some NP Problems

- The following problems, among others, can be solved nondeterministically in polynomial time:
  - The Satisfiability problem
  - The Hamiltonian path problem: given an undirected graph with n vertices, find a simple path that passes through all the vertices
  - The Clique problem: given an undirected graph with n vertices, find a subset of k vertices such that there is an edge between every pair of vertices in the subset
- These problems have deterministic solutions with exponential time complexity, but no deterministic polynomial solution has been found

Polynomial-Time Reduction

- Since NP problems have similar characteristics, it is convenient to determine if they can be reduced to each other
- A language L_1 is *polynomial-time reducible* to another language L_2 if there exists a deterministic Turing machine that can transform any string w_1 in L_1 to a string w_2 in L_2 so that
  - The transformation can be completed in polynomial time, and
  - w_1 is in L_1 if and only if w_2 is in L_2
- Consider 3SAT, a modified version of the SAT problem in which each clause can have at most three literals; as shown in Examples 14.9 and 14.10,
  - The SAT problem is polynomial-time reducible to 3SAT
  - The 3SAT problem is polynomial-time reducible to CLIQUE

Example 14.9: The SAT problem is polynomial-time reducible to 3SAT

- We illustrate the reduction with the simple 4-literal expression

$$e_1 = (x_1 \vee x_2 \vee x_3 \vee x_4)$$

- We introduce a new variable z and construct

$$e_2 = (x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \overline{z})$$

- Now, we can see

$$e_2 = \begin{cases} x_1 \vee x_2 & if\ z = 0 \\ x_3 \vee x_4 & if\ z = 1 \end{cases}$$

- and

$$e_1 = 1 \Leftrightarrow x_1 \vee x_2 = 1 \text{ or } x_3 \vee x_4 = 1$$

- So

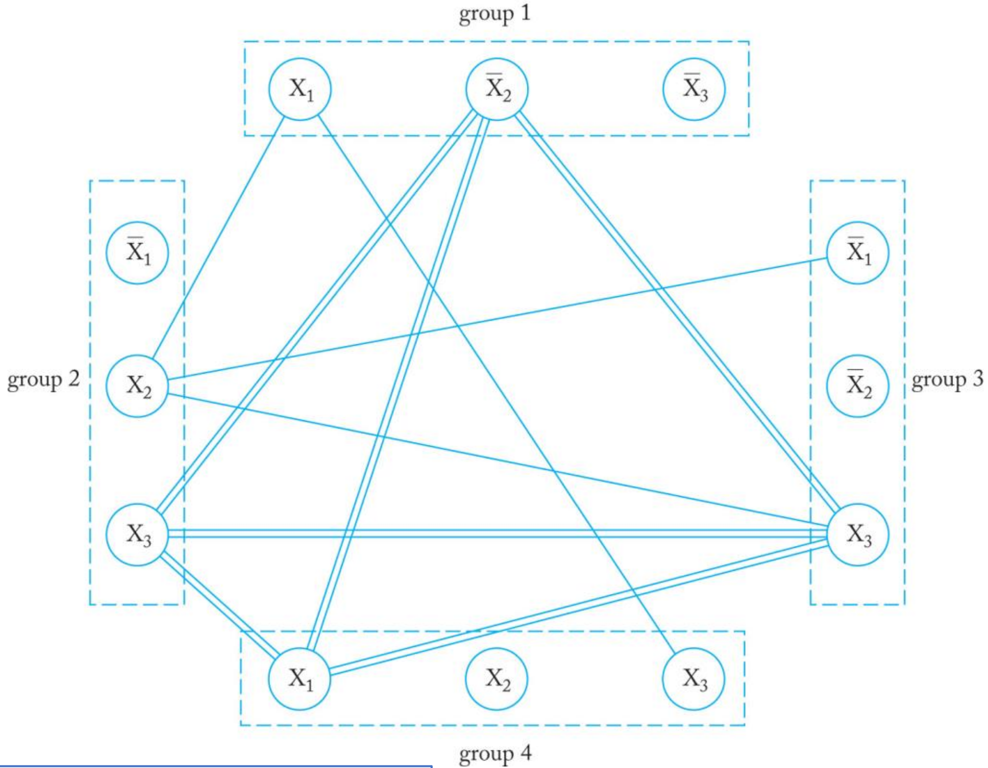
$$e_1 = 1 \Leftrightarrow e_2 = 1$$

Example 14.10: The 3SAT problem is polynomial-time reducible to CLIQUE

- Assume that in any 3SAT expression each clause has exactly three literals. Consider an expression

$$e = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

- Draw a graph for the expression *e*.
  - Each clause is represented by a group of three vertices
  - Each literal is associated with one of the vertices
  - For each vertex in a group, put in an edge to all vertices of the other groups, unless the two associated literals are complements



the subgraph with vertices $(\overline{x_2})_1$, $(x_3)_2$, $(x_3)_3$, $(x_1)_4$, is a 4-clique

$\overline{x_2} = x_3 = x_1 = 1$ is a variable assignment that satisfies e.

NP-Completeness

- Some problems have been identified as being as complex as any other problem in NP
- A language (or problem) L is *NP-complete* if
  - L is in NP, and
  - Every problem in NP is polynomial-time reducible to L
- As stated in Theorem 14.5, the Satisfiability Problem is NP-complete
- This definition is very significant because, if a deterministic polynomial-time algorithm is found for any NP-complete problem, then every language in NP is also in P

An Open Question: P = NP?

- Computer scientists continue to look for an efficient (deterministic, polynomial-time) algorithm that can be applied to all NP problems, therefore concluding that P = NP
- On the other hand, if a proof is found that any of the NP-complete problems is intractable, then we can conclude that $P \subset NP$ and that many interesting problems are not practically solvable
- In spite of our best efforts, no efficient algorithm has been found for any NP-complete problem, so our conjecture is that $P \neq NP$
- However, until a proof is found, *P = NP? remains the fundamental open question in complexity theory*