

Formal Languages

Chapter 14 An Introduction to Computational Complexity

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

September 15, 2008

Chapter Outline

1. Efficiency of Computation
2. Turing Machines 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

Previously, we studied whether a function is *computable*. In practice, some computable functions are not *feasible* in that it may take millions of years or need un-imaginable amount of space. These functions are deemed *intractable*.

Complexity of software in the broad sense includes many factors: time for designing/writing/testing the program, time for compiling the program, time for executing the program, time for maintenance, personnel required, space for execution, reliability of the program, etc.

In the *complexity theory*, we are concerned with two factors:

- time complexity
- space complexity

We will concentrate on time complexity in this chapter.

§14.1 Efficiency of Computation

Example. Consider a concrete example: we want to sort 1,000,000 integers in increasing order. How much time do we need?

The actual time depends on many factors.

The actual time depends on the computer we are using. A simple computer may take 5 minutes while a supercomputer may take only 10 seconds. In order for our study to be independent of the particular computer used, we count the number of steps a machine may take, instead of measuring the actual execution time.

Different computers may also take different numbers of steps. For instance, a Turing machine with 1 tape may take more steps than a Turing machine with 2 tapes. Here we will assume a multi-tape Turing machine.

There are many different sorting algorithms. Some are quicker in some situations, others are quicker in other situations. Thus, the

actual number of steps is also dependent on the algorithms. We will analyze each algorithm in turn and look for the *best* algorithm for a problem.

If the 1,000,000 integers are almost sorted initially, sorting them may take less number of steps. On the other hand, if the 1,000,000 integers are quite random initially, sorting them may take more steps. This means the actual time may vary according to the initial condition. Here we are interested in the *worst-case* time.

The actual number of steps is also affected slightly by the programmer's expertise and is hard to count. Here we are satisfied with the *approximate* counts, which is called the *asymptotic complexity*, using the notations O , Ω and Θ .

The complexity of an algorithm or a problem is a function of the *size* of the problem, written as $f(n)$.

Example 14.1. Consider a set of n numbers x_1, x_2, \dots, x_n . We want to determine if the set contain a specific number x .

The size of this instance of the problem is n , the number of items in the set.

A simplest algorithm is a linear search, comparing x against each of x_1, x_2, \dots, x_n in turn. In the best case, one comparison suffices. In the worst case, we need n comparisons. We say the complexity of this linear search is $O(n)$, or even better $\Theta(n)$.

§14.2 Turing Machines and Complexity

In terms of *computability*, all kinds of Turing machines are equivalent in that a problem that can be solved by one kind of TM can also be solved by other kinds of TMs.

In terms of *complexity*, the Turing machines are NOT equivalent. Different TMs may take different numbers of steps for the same problem.

Example 14.2. In example 9.4, we construct a 1-tape TM for the language: $\{a^n b^n \mid n \geq 1\}$. For a specific string, say $a^n b^n$, the TM takes roughly $2n$ steps in matching each a with the corresponding b . The complexity on this 1-tape TM is $O(n^2)$.

Now consider a 2-tape TM. We first copy the a^n part to the 2nd tape and then match a 's on the 2nd tape with b 's on the 1st tape. This approach (on the 2-tape TM) takes $O(n)$ time.

Example 14.3. (Satisfiability Problem) A boolean expression, such as $(x \vee y) \wedge ((z \vee \neg x) \vee (\neg y \wedge \neg x))$, is made up of the variables x, y, z , etc., and boolean operators \vee, \wedge, \neg . The variables may take values *true* or *false*. The boolean operators are defined in the usual way. A boolean expression is *satisfiable* if and only if there is an assignment of truth values to the variables so that the whole expression is *true*. We are confined to consider only boolean expressions in the conjunctive normal form, which is a conjunction of the disjunctions of variables and the negation of variables.

An example of a conjunctive normal form:

$$(x \vee y \vee \neg u) \wedge (\neg y \vee \neg z \vee w) \wedge (\neg u \vee w \vee p)$$

This naive-looking satisfiability problem is quite interesting in the study of the complexity theory. A simple, deterministic method is to try each of the 2^n possibilities, where n is the number of variables. The complexity of this exhaustive method is thus 2^n .

Surprisingly, up to now, nobody knows a better deterministic algorithm for this satisfiability problem.

A *non-deterministic* method simply guesses the appropriate values for the variables in order to satisfy a boolean expression. This nondeterministic method takes time $O(n)$.

We will assume a multi-tape TM in later discussion.

§14.3 Language Families and Complexity Classes

Chomsky hierarchy is a classification of languages. We could also use time complexity to classify languages.

Definition 14.1. A Turing machine M accepts a language L in time $T(n)$ if and only if every string $w \in L$ is accepted by M in $O(T(|w|))$ moves. When M is nondeterministic, there is a sequence of moves (whose length is $O(T(|w|))$) that leads to acceptance.

Definition 14.2. A language L belong to the class $DTIME(T(n))$ if and only if there is a deterministic TM that accepts L in time $T(n)$. A language L belong to the class $NTIME(T(n))$ if and only if there is a nondeterministic TM that accepts L in time $T(n)$.

Some simple relations among the complexity class are

- $DTIME(T(n)) \subseteq NTIME(T(n))$
- $T_1(n) = O(T_2(n))$ implies $DTIME(T_1(n)) \subseteq DTIME(T_2(n))$.

Theorem 14.1. For every $k \geq 1$,

$$DTIME(n^k) \subsetneq DTIME(n^{k+1})$$

Theorem 14.2. There is no total Turing-computable function $f(n)$ such that every recursive language is in $DTIME(f(n))$.

Proof. Consider the alphabet $\Sigma =_{def} \{0, 1\}$, with all strings in Σ^+ arranged in some order w_1, w_2, w_3, \dots . Also assume the collection of all TMs are arranged in some order M_1, M_2, M_3, \dots . Assume the function f stated in the theorem does exist. We will derive a contradiction.

First define the language

$$L =_{def} \{w_i \mid w_i \text{ is not accepted by } M_i \text{ in } f(|w_i|) \text{ steps} \}$$

We claim that L is recursive. To see this, consider any $w_i \in L$ and compute first $f(|w_i|)$. Because f is assumed to be total and Turing-computable, this is possible. We run

M_i on w_i for $f(|w_i|)$ steps. This will determine if $w_i \in L$. Hence, L is recursive.

Next we show that $L \notin DIMTE(f(n))$. Since L is recursive, $L = L(M_k)$ for some Turing machine M_k . Now consider if $w_k \in L$. There are two cases.

- Assume $w_k \in L$. Since $L \in DTIME(f(n))$, M_k accepts w_k in $f(|w_k|)$ steps. However, according to the definition of L , $w_k \notin L$. This is a contradiction.
- Assume $w_k \notin L$. Since $L \notin DTIME(f(n))$, M_k does not accept w_k in $f(|w_k|)$ steps. However, according to the definition of L , $w_k \in L$. This is also a contradiction.

Either case leads to a contradiction. We conclude that there is no such f . \square

From Theorems 14.1 and 14.2, we may derive several (insignificant) corollaries.

Example 14.5. Every regular language can be recognized by a deterministic finite automaton in linear time. Let L_{REG} denote the class of all regular languages. We have

$$L_{REG} \subseteq DTIME(n)$$

However, many context-free languages, such as LL(1), LR(1), $\{a^n b^n \mid n \geq 0\}$, etc., are also in $DTIME(n)$.

Example 14.6. The context-free language
 $L =_{def} \{ww \mid w \in \{a, b\}^*\}$ is in $NTIME(n)$.

The method is as follows: Nondeterministically determine the middle of the input. Copy the first half to the 2nd tape. Then compare the contents of the two halves. This method takes $O(|w|)$ steps. Thus, $L \in NTIME(n)$.

Actually, $L \in DTIME(n)$. We can decide the middle of the input by counting. (Consider a 2-tape TM.)

Example 14.7. Due to the CYK parser, we know the class of all context-free language $L_{CF} \subseteq DTIME(n^3)$ and $L_{CF} \subseteq NTIME(n)$ (the size of a syntax tree is proportional to the length of the input).

For context-sensitive languages, we can use an exhaustive search method for membership testing. This could be done because only a finite number of productions are applicable at each step. Thus, every context-sensitive language belongs to some $DTIME(n^M)$, where M is determined from the grammar. (But this does not mean that $L_{CS} \subseteq DTIME(n^M)$.)

§14.4 The Complexity Classes P and NP

Chomsky's hierarchy is a classification of languages.

Alternatively, we could classify languages according to their time complexity, such as $DTIME(n^k)$, $DTIME(n^{k+1})$, etc.

The difference between some classes, say $DTIME(n^k)$ and $DTIME(n^{k+1})$, is not significant since this difference might be the result of the particular TM model we adopt. It is not clear what an ideal model should.

For this reason, we may define the famous class

$$P =_{def} \bigcup_{i \geq 1} DTIME(n^i)$$

P include L_{REG} and L_{CF} .

But the difference between some other classes, say $DTIME(n^k)$ and $DTIME(e^n)$, is quite significant. The difference seems to be

rooted in the nature, not the particular model, of the problem.

We feel that the difference between *deterministic* and *nondeterministic* complexity classes is significant, we define

$$NP =_{def} \bigcup_{i \geq 1} NTIME(n^i)$$

Obviously, $P \subseteq NP$. One of the major current challenge in computer science is to decide if $P = NP$, that is, if there is a language that is in NP but not in P .

Most people believe that $P \neq NP$ but none can come up with a convincing proof.

Beyond decidability problems, we are also interested in knowing if a language is *practically feasible*, that is, if there is a practical algorithm for that language.

For example, the best known algorithm for the satisfiability problem of boolean expressions takes exponential time. The problem is deemed *intractable* because, even for an instance of a small size, the algorithm will need an unreasonable amount of time.

Cook-Karp Thesis. Problems in P are *tractable* while all other problems, including those in NP , though decidable, are *intractable*.

Most people agree that problems not in P demand more resource (that is, time) than is reasonable or available.

But are problems in $DTIME(n^{100})$ tractable?

Empirically, most practical problems in P are in $DTIME(n)$, $DTIME(n^2)$, or $DTIME(n^3)$ while those outside P tends to have exponential complexity. Few problems are in, say $DTIME(n^{100})$. A clear distinction exists between problems in P and those not in P .

Definition. $\text{complexity}(\text{algorithm } A)$ is the worst-case time requirement of algorithm A , that is, for all possible input.

Definition. $\text{complexity}(\text{problem } P/\text{language } L)$ is the time requirement of the best algorithms for solving problem P (or accepting language L).

Currently the most challenging problem is if $N = NP$.

Definition. A language L_1 is *polynomial-time reducible* to another language L_2 if there is a deterministic TM by which every string w_1 over Σ_1 can be transformed into a string w_2 over Σ_2 such that $[w_1 \in L_1$ if and only if $w_2 \in L_2]$.

If L_1 is polynomial-time reducible to L_2 , then the complexity of L_1 is no more than that of L_2 (plus a polynomial factor).

We use the notation

$$L_1 \sqsubseteq_f L_2 \text{ if and only if } \text{complexity}(L_1) \leq \text{complexity}(L_2) + f$$

Corollary. Suppose L_1 is polynomial-time reducible to L_2 . Then if $L_2 \in P$ then $L_1 \in P$. Similarly, if $L_2 \in NP$ then $L_1 \in NP$.

NP-Completeness

Definition. A language L is *NP-complete* if $L \in NP$ and every language $L' \in NP$ is polynomial-time reducible to L .

All NP-complete languages can be mutually polynomial-time reducible to one another. In this sense, they have the *same* time complexity. If any NP-complete language has a polynomial-time algorithm, so do all NP-complete languages and all NP languages.

Therefore, NP-complete problems are the *hardest* problems in NP.

Example. This is an NPC problem: “is there a Hamiltonian path (a path that goes through every vertex exactly once) in a graph?”

Corollary. Suppose (1) L_1 is polynomial-time reducible to L_2 and (2) $L_2 \in NP$. Then if L_1 is NP-complete, so is L_2 .

Discussion. (1) The assumption that L_1 is NP-complete implies that L_2 is polynomial-time reducible to L_2 . Since L_1 and L_2 are mutually polynomial-time reducible to each other, they should belong to the *same* class.

(2) If $L_2 \in P$, then $L_1 \in P$. This together with the assumption that L_1 is NP-complete implies that $P = NP$, which is an open question currently. \square

We used this “*polynomial-time reduction*” technique to prove many problems are NP-complete. See the book: Garey and Johnson, “Computers and Intractability,” 1979.

However, what about the *first* NP-complete problem? The first NP-complete problem is the satisfiability problem, proved by Cook in 1971.

Example. The satisfiability problem can be viewed as a language problem. We can encode a boolean expression e as a string w_e over some alphabet and ask for a Turing machine M such that [e is satisfiable if and only if M accepts w_e].

Many problems can be similarly translated into corresponding language problems. The complexity of the original problem is defined as the complexity of the corresponding language problem.

Cook Theorem. The satisfiability problem is NP-complete.

To prove another problem Q is NP-complete, we simply polynomial-time-reduce the satisfiability problem into Q directly or indirectly. In this way, many problems have been proved NP-complete.

$$SAT \sqsubseteq_{\text{polynomial-time}} Q$$

However, up to now, nobody has come up with a polynomial-time

algorithm for any of these NP-complete problems. This makes most people *believe* that $P \neq NP$.

A sample of NPC problems:

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

Theorem. CSAT is NP-complete.

Theorem. 3SAT is NP-complete.

Theorem. The independent-set (IS) problem is NP-complete.

Theorem. The node-cover (NC) problem is NP-complete.

Theorem. The Directed Hamilton-Circuit problem is NP-complete.

Theorem. The Undirected Hamilton-Circuit problem is NP-complete.

Theorem. TSP is NP-complete.

Review.

A little history:

- Turing machines, 1936.
- Finite automata, 1940's-1950's.
- Chomsky's grammars, 1950's.
- Cook's tractability theorem, 1971.

Turing machines are similar to Darwin's Evolution Theory in that they both bear much influence outside their original domains.

Turing machines influence the limits of human thinking and the study of machine intelligence. Darwin's theory carries much social implication.

[

Index

]

complexity theory, 3

NP-complete language, 22

polynomial-time reduction, 21

satisfiability problem, 8