NP-Completeness: an Overview

Young Eun Kim and Gene Moo Lee
Department of Computer Science and Engineering, Korea University

Abstract
This paper presents an overview of NP-complete problems. The theory of NP-completeness is important not only in theoretical aspect but also in reality. First, we will take a look at the formal definition and some examples of NP-complete problems. Then, we will see how to prove a problem is NP-complete and how to cope with NP-complete problems.

Keywords – NP-completeness, class P, class NP, nondeterministic Turing machine, polynomial-time reducibility

Contents

1. Introduction

IN the theory of computation, there are two major approaches. First, we want to know what kind of problems can be solved in the computer (Computability Theory). If we could solve the problem, secondly, then we want to know how fast the problem can be solved (Complexity Theory). In the Complexity Theory, we analyze the algorithm in two points of view: time and space. In the theory of NP-completeness, we categorize the problems by the time complexity.

2. Motivation

So far in this class, we've learned many algorithms to solve problems like Sorting, Shortest-Path, String Matching, etc. And the time complexity of those algorithms was all in the form O(n), $O(n^2)$, or O(n*log n). But in the reality, not all problems have polynomial time algorithm. In some situation, we have to check out all the possible situations to find the correct answer, and in that case, we could have an exponential time algorithm like of $O(2^n)$ time complexity.

But the problem is that these exponential time algorithms may be just useless. [Table 1] indicates approximated computing time in the given time complexity, and shows that it is impossible to compute exponential time.

	10	30	60
O(n)	0.00001 sec	.00003 sec	.00006 sec
O(n^2)	.0001 sec	.0009 sec	.0036 sec
O(n^3)	.001 sec	.027 sec	.216 sec
O(n^5)	1 sec	24.3 sec	13.0 min
O(2^n)	.001 sec	17.9 min	**366 centuries**
O(3^n)	.059 sec	6.5 yrs	**1.3*10^13 centuries**

[Table 1] Polynomial and exponential time complexity

To solve Traveling Salesperson Problem (TSP) in brute-force algorithm in the case "n = 1000", we have to compute 1000! cases. However, even if all the electrons in the universe have computing power of super-computers and they work from the beginning of the universe, we can't compute all 1000! times. In other words, it is simply impossible to do solve that problem!

In reality, we faced many problems like TSP, which is called to be ***NP-complete***. In this paper, we present the definition of NP-complete problems first. Then we show how to prove a problem is NP-complete and how to cope with NP-complete problems.

3. Background Knowledge

Before defining NP-completeness formally, we have to define three notations: class P, class NP, and polynomial time reducibility.

3.1. Decision and Optimization Problems

Many problems of interest are ***optimization problems***, in which each feasible (i.e., "legal") solution has an associated value, and we wish to find the feasible solution with the best value. However, NP-completeness applies directly not to optimization problems, but to ***decision problems***, in which the answer is simply "yes" or "no" Then we can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

For example, in a problem that we call Shortest-Path, we wish to find the path from *u* to *v* that uses the fewest edges. The related decision problem, which we call PATH, is whether the given graph has a path from *u* to *v* consisting of at most *k* edges. Thus, even though the theory of NP-completeness restricts its attention to decision problems, the theory often has implications for optimization problems. From now, we will consider decision problems.

3.2. Turing Machine and class P

In order to know the exact ability of computers, we have to define the mathematical model of real computers. By Church-Turing thesis, we accept that the power of real computers is equivalent to that of Turing machines. So we can know the power of computers by analyzing equivalent Turing machines. The formal definition of Turing machine is the following

Definition 1

A ***Turing machine*** is a 7-tuple ($Q, \sum, \Gamma, \delta, q_0, q_{accept}, q_{reject}$), where,

1) Q is the finite set of states,
2) ∑ is the finite set of input alphabets,
3) Γ is the tape alphabet, where ⌂ Γ and ∑ Γ,
4) $\delta : Q*\Gamma \rightarrow Q*\Gamma*\{L,R\}$ is the transition function,
5) q_0 Q is the start state,
6) q_{accept} Q is the accept state, and
7) q_{reject} Q is the reject state, where $q_{accept} \neq q_{reject}$.

Now, we can define the class of problems that have efficient algorithms to resolve, namely class P.

Definition 2

P is the class of languages that are decidable in polynomial time on a Turing machine. In other words,

$$P = \ _{k>0} \text{TIME}(n^k).$$

Then problem like Sorting is in P because it has algorithms with time complexity $O(n^2)$. The class P plays a central role in our theory and is important because 1) P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape

Turing machine, and 2) P roughly corresponds to the class of problems that are realistically solvable on a computer.

3.3. Nondeterministic Turing machine and class NP

Unlike those problems in P, we don't know of a fast way to determine whether a graph has a Hamiltonian path (HAMPATH problem). However, if such a path were discovered somehow, we could easily convince that the path is Hamiltonian. In other words, *verifying* the existence of Hamiltonian path may be easier than *determining* its existence. This kind of problem is in class NP.

Definition 3

NP is the class of languages that have polynomial time verifiers.

We could also define the class NP with a powerful version of Turing machine which has guessing ability. The following is the formal definition of this machine.

Definition 4

A ***nondeterministic Turing machine*** is a Turing machine with the transition function has the form

$$\delta : Q * \Gamma \rightarrow P(Q * \Gamma * \{L, R\}).$$

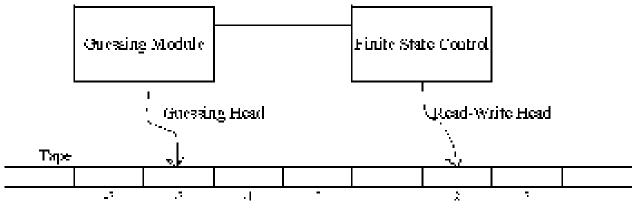
The conceptual diagram of nondeterministic Turing machine (NTM) is in [Figure 1].

The power of an NTM is the same as that of deterministic Turing machine *in view of computability*. But *in view of complexity*, NTM can solve problems much faster than a normal Turing machine. In addition, NTM has a guessing module to choose the evidence answer nondeterministically.

We can determine whether a problem is in NP or not by the following theorem. In fact, the class NP is defined in terms of NTM in some books.

Theorem 1

A language is in NP if and only if it is decided by some nondeterministic polynomial time Turing machine.



[Figure 1] Schematic representation of NTM

We can describe the two classes above like this.

P = the class of languages where membership can be ***decided*** quickly.

NP = the class of languages where membership can be ***verified*** quickly.

3.4. Polynomial-time Reducibility

The last thing to understand the notion of NP-completeness is polynomial-time reducibility.

Definition 5

Language A is ***polynomial time reducible***, to language B, written A B, if a polynomial time computable function f : ∑* → ∑* exists, where for every w ∑*,

w A iff f(w) B.

The function f is called the ***polynomial time reduction*** of A to B.

Let's see the underlying meaning of this definition. If we have a polynomial time reduction from A to B, then it means that problem A can be converted to problem B, and problem B is "no harder to solve" than problem A. In other words, A is harder than B, or they are equally hard to solve.

4. Definition of NP-completeness

Now, we can define NP-completeness formally.

Definition 6

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. Every A in NP is polynomial time reducible to B.

A language is said to be ***NP-hard*** if it satisfies the second condition above.

NP-Complete problem is the "hardest" problem in class NP because all problems in NP can be transformed to the problem in class NP-complete. Also, all NP-complete problems are *unknown* whether a polynomial-time algorithm exists.

To explain the underlying meaning, "NP" means **N**ondeterministic **P**olynomial. We add the word "complete" because **if one** of NP-complete problems is proved to be solved in polynomial time, then it means that we can solve **all** the NP-complete problems in polynomial time.

5. Some Examples

Let's look at some examples of NP-complete problems.

- Satisfiability: Given a propositional formula φ, is there a truth assignment that makes φ to be true?
- Traveling Salesperson Problem: Given *n* cities and roads between the cities, what is the path to visit each city one time with minimum cost?
- Longest Path: Given a graph and two vertices *s* and *t*, what is the longest path from *s* to *t*? (cf. Shortest Path is in P)
- Real-time Scheduling: Given a set of processes with release time and deadline, is there a schedule to satisfy the release time constraints and to meet all the deadlines?
- Hamiltonian Cycle: Given a directed graph, does the graph have a Hamiltonian cycle? (cf. Euler Cycle is in P)

6. Hierarchy of Problems

In this chapter, we want to know the locations of P and NP problems in the whole hierarchy of problems.

6.1. Undecidable Problems

In the past, people thought that computers can do anything with enough amounts of memory and time. However, it is proved that computers cannot solve some problems. Those problems are called ***undecidable***. For instance, Halting problem, determining whether a Turing machine halts (accepts or rejects) on a given input, is undecidable.

6.2. P, NP, PSPACE, and EXPTIME

Before explaining the relationships, we had better know the definition of P, NP, PSPACE, and EXPTIME.

1) **P** is the class of languages that are decidable in polynomial time on a Turing machine.

2) **NP** is the class of languages that have polynomial time verifiers.

3) **PSPACE** is the class of languages that are decidable in polynomial space on a deterministic Turing machine.

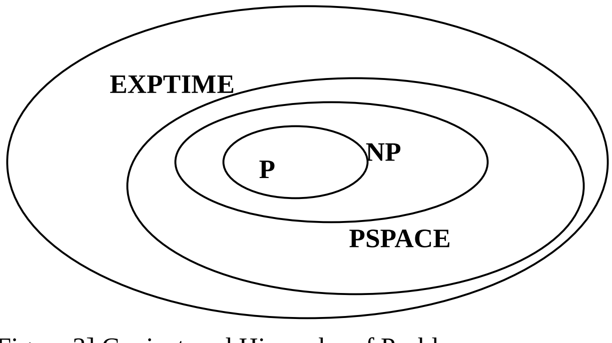
4) **NPSPACE** is the class of languages that are decidable in polynomial space on a nondeterministic Turing machine.

5) **EXPTIME** is the class in which some TM decides problems and halts in exponential time.

The relationship among P, NP, PSPACE, and EXPTIME is like this:

P   NP   PSPACE = NPSPACE   EXPTIME

We don't know whether any of these containments is actually equality. We may not yet discover a simulation about these relationships. However, P ≠ EXPITME has been proved. Therefore at least one of the preceding containment is proper, but we are unable to say which! Indeed, most researchers *believe* that all the containments are proper.



[Figure 3] Conjectured Hierarchy of Problems

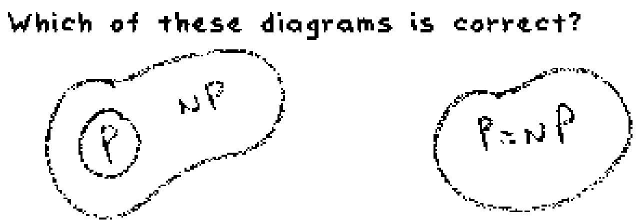
6.3. Conjecture with P and NP problems

1) By general definition, **P   NP**. This means that an efficient algorithm on a deterministic machine ***does not exist*** relevant to a NP problem.

2) On the contrary, another relation, **P = NP**, is included in the list of the most important mathematical problems for the

new century. P = NP means that an efficient algorithm on a deterministic machine ***is not found yet*** relevant to a NP problem.

Figure 2 represents the conjectures with P and NP problems.



[Figure 2] Two Conjectures between P and NP

7. How to prove a problem is NP-complete

Now, let's see how to prove a hard problem is NP-complete. First, let's see the following theorem.

7.1. Proving NP-Completeness

<u>Theorem 2</u>

If a problem C is in NP, and there exists NP-Complete problem B with B   C, then C is NP-complete.

Since all problems in NP can be polynomial time reduced to B and B can be reduced to C, we can conclude that all problems in NP can be polynomial time reduced to C.

Then to prove a problem is NP-Complete, we need at least one NP-complete problem. By the following theorem, we get the first NP-complete problem.

7.2. Cook's Theorem

<u>Theorem 3 (Cook)</u>

Satisfiability (SAT) is NP-complete.

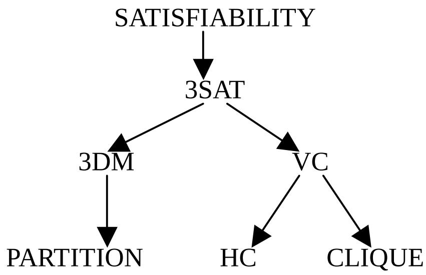
Recall that Satisfiability problem (SAT) is to test whether a Boolean formula is satisfiable.

SAT = {<Φ> | Φ is a satisfiable Boolean formula}

A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

To prove this theorem we need five pages of space!! If you want to see it, please look at reference [3] or [4].

7.3. NP-complete problems Tree



[Figure 3] Polynomial Reductions with six problems

It can be shown easily that these six problems (3SAT, 3DM, VC, PARTITION, HC, and CLIQUE) are NP problem. First, there problems are in NP because a nondeterministic algorithm can guess a truth assignment for the variables and check in polynomial time whether that truth setting satisfies whole the problems (L   NP). Also, by the reduction in Figure 3, we can show that all NP problems can be reduced to these problems. Therefore, they are NP-Complete.

8. How to cope with NP-complete problems

Even though we proved that a problem is NP-complete, the problem will not disappear and we have to find alternatives to resolve the problem. Here we propose three ways to cope with NP-complete problems.

8.1. Using Heuristic Algorithms

Heuristic algorithm is *to find a "good" solution within an acceptable amount of time*, not to find the best solution. The most widely applied technique is that of "neighborhood search" (or local search). In this technique, a pre-selected set of local operations is used to repeatedly improve an initial solution. This process is continued until no further local improvements can be made and a "locally optimum" solution has been obtained. In other words, heuristic algorithm reduces the search space. For example, there are practical solutions to solve SAT problem: zChaff, Berkmin, GRASP, SATO, QingTing, etc. These solvers solve SAT in considerable amount of time.

8.2. Approximation

An approximation algorithm is designed to find ***approximately optimal*** solutions. In practice, we may not need the absolute best or *optimal* solution to a problem. A *nearly optimal* solution may be good enough and may be much easier to find.

For example, there is an approximation algorithm for finding the smallest vertex covers. This algorithm produces a vertex cover that is *never more than twice the size* of one of the smallest vertex covers.

8.3. Quantum Computation

Quantum computing uses a pulse of light. In detail, the bits, 0 and 1, are substituted by the spins of quantum. The spin is expressed by 'spin up' (0) or 'spin down' (1). Because the speed of processing by quantum is equal to that of light, quantum computer can calculate a time-consuming problem in a relatively short time than current computers. Therefore, some NP problems requiring about 300 years or more to find an answer can be solved just in a few seconds. For instance, an algorithm was developed for factoring numbers on a quantum computer which runs in $O((\log N)^{2+\varepsilon})$ steps. This is roughly quadratic in the input size, so factoring a 1000 digit number with such an algorithm would require only a few million steps.

9. Summary

Theoretically, the class NP-complete is important because if one of the problems in NP-complete is proved to have a polynomial time algorithm, then we have polynomial algorithm to solve all the problems in NP. Also, if we find such an efficient algorithm, then the class P and NP become the same. However, we believe that those two classes are different, i.e, P is a proper subset of NP.

Practically, as we saw in this book, we know that many real problems of our interest are NP-complete, and to be NP-Complete means that many scholars and researchers have tried to solve the problem in efficient algorithm but failed to achieve such a good algorithm. We can prove that a problem is NP-complete, just by finding a polynomial time reduction from an existing NP-complete problem to the would-be-proved problem as NP-complete. After proving, we don't have to struggle to find such an efficient algorithm, instead we could solve the problem in different ways: 1) we can use heuristic algorithms to reduce the search space, 2) we may try to find approximated results instead of the exact answers, and 3) in the future, we could apply quantum computation to solve the problems.

References

[1] , . C .

[2] Cormen, Leiserson, Rivest. *Introduction to Algorithms*.

[3] Sipser. *Introduction to the Theory of Computation*.

[4] Garey, Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness*.

[5] Kitaev, Shen. *Classical and Quantum Computation*.

Epilogue

Young Eun Kim ( ) student#: 2000160129

019-302-9214 / gimyoungeun@hotmail.com

It was a good experience to prepare for the presentation. During preparing, I got chances to study about a given subject in detail, search for information through many books and websites, and discuss with my partner, Gene Moo. I appreciate for Prof. Hwang to give me the good opportunity.

Gene Moo Lee ( ) student#: 2000160184

011-9931-2518 / gm0707@hanmail.net

http://klug.korea.ac.kr/~gm0707

I am preparing for studying abroad and my main interest is the theory of computation. So it was good time for me to prepare this subject. I really want everybody come to understand by our presentation. If you have any curiosity, feel free to contact me. I want to thank Young-Eun for collaboration and thank Prof. Hwang to give us such a good opportunity.