A Review on Computational Complexity Theory and Classes of Problems

Hussain Ahmad
Higher Education Department (Commerce Wing). KPK Peshawar, +923469361048,
hussainternow@gmail.com

Abstract: There are various classes of problems in complexity theory also some mathematical precisely defined problems that cannot be solved by algorithms even for unlimited time. Yuri Matiyasevich and Alan Turing proved that no such algorithms exist to solve the halting problems. Turing machines also called computing machines are not algorithms but they provide mathematical definition of objects so that one can prove their existence or none-existence. The unknown status of NP Complete problems is another failure story. Till now there is no algorithm discovered that can solve the NP complete problem in polynomial time also nobody has been able to prove that no polynomial-time algorithm exists for any of them. The interesting part is, if any NP complete is solved in polynomial time, and then all the NP complete problems can be solved in polynomial. In this paper we present a review on complexity theory, various problems' classes etc. Then will discuss various terms related to complexity theory such as decision problems, function problems, Polynomial Problems, Exponential Problems, NP, NP Hard and NP-complete.

Keywords: Class P, NP-completeness, polynomial, algorithm, optimization.

Introduction

There are enormous computations methods are being used to solve complex problems such as Minimum spanning Tree, Euler graph, shortest path, and so on. But is there solution exist for all mathematical problems? The answer is no. There are various mathematical problems such as analysis, logic, set theories and some diophantine equations which cannot be solved algorithm by even with unlimited time also called Hilbert's Tenth problems. Similarly the status of class NP and NP complete problems is unknown.

In this paper we will discuss computational complexity theory, various classes of computation problems, their properties and their relation. Then we will specifically give introduction to the class P, Exponential Problems, NP and NP complete problems. Then we will investigate computation model for each class, examples of each class problems. We will also discuss various applications and importance of these problems in computation theory.

Computational Complexity Theory

In computer science theory, the computational complexity theory is that branch which classifies problems into various classes based on their difficulty and relation among them. By using whatever algorithm, the difficulty of a problem is inherently measured based on resource required for solution [16]. Times, space, amount of communication and so on are the resources needed to find problems solutions used by algorithm. Based on complexity of the problem, we define complexity classes [16]. In this section we describe various complexity classes.

Decision problems

These problems are also called formal languages. In complexity theory, a decision problem is a special problem which gives yes or no. The objective is to decide, if an input string is in the language and decider algorithm return yes, the algorithm accept it otherwise reject the input string [16].

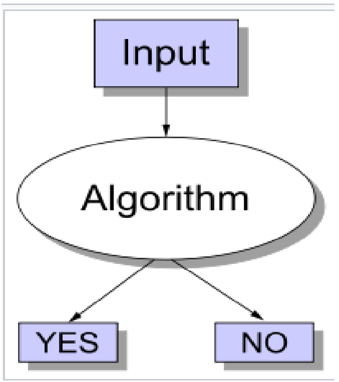


Figure 01: Decision Problem's Algorithm

A. Function problems

These are computational problems where for every input, a single output is expected and the output is more complex than decision problem's output i.e. the output is not only *yes* or *no*. Examples include traveling salesman problem and the number factorization problem.

B. Polynomial problem (p)

It contains all those languages or problems that we can solve in reasonable amount of time or in practical time actually in polynomial time. If an algorithm solve a problem in polynomial time i.e. T (n) = $O(n^c)$, then it is called class P, PTIME or DTIME($n^{O(1)}$) problems (polynomial problem) where c is a positive constant. Formal definition of class P problem is:

Def: A language L1 is in class P if and only if there exist a Deterministic Turing Machine (DTM) M, such that [16]

- "M runs for polynomial time on all inputs
- For all x in L, M outputs 1
- For all x not in L, M outputs 0"

Examples include the following:

a. Binary search tree = O(n log n)
b. Depth-first search/Breath-first search.
c. Sorting: O(n log n) = $O(n^2)$
d. Calculating maximum matching.
e. All-pairs shortest path: $O(n^3)$
f. Finding greatest common divisor.
g. Minimum spanning tree: (E log E)= $O(E^2)$

Consider we have a list of integers and we want to display the smallest number in the list using an algorithm. One method is the iteration, examine the entire integer in the list and keep track of the smallest number we have seen up to that point. Every time we look at number, we compare it to the current smallest number, and if it is smaller, we update the smallest number variable. What will be computation time? If there are m elements in the list, the algorithm performs a constant number of operation i.e. the algorithm runs in O(m) time, or that the running time is a linear function. So this algorithm runs in linear time. Similarly there are various algorithms that required quadratic $O(n^2)$, or exponential $O(2^n)$ or even logarithmic time (O(log n)) and so many.

In short, if the computation time of an algorithm to solve a problem is, for instance linear time or quadratic time or cubic time, then we can say that problem is in class P, because time required for algorithms decision is polynomial time [1].

D. Exponential problems (e)

Exponential problems are those if it is impossible to develop polynomial-time algorithms for them. And an algorithm solve in O ($n^{v(n)}$), where if n goes to infinity then v(n) also goes to infinity. In exponential context we can divide computation problems into the following classes:

a. Polynomial (P)
b. Exponential (E)
c. Undecidable or Intractable (I)

Now we turn to very important and large class of computational complexity theory, the class NP. By analyzing, we can conclude the following.

1. How the problems solve exponentially,
2. We don't know how will the problems solve in polynomial time, and
3. We are not sure if the problems can be solved in polynomial time at all.

The Nondeterministically Polynomial (Class NP) is a gray/unexplored area between class P and the class E. We discuss it in the following section.

E. Nondeterministically Polynomial problems (np)

- *Definition No. 1 of NP*

If there exist a nondeterministic Turing Machine and it can solve the problem in a polynomial number of nondeterministic moves then the problem is said to be class NP problems. A language 'L' is in class NP if a nondeterministic Turing machine M' decides it in polynomial time that is M' runs in dn^k.

- *Definition No. 2 of NP*

A class NP problem is a problem whose solution belongs to a finite set of possibilities and to find the correctness of candidate solution, polynomial time is required. Clearly, the class P is the subset of NP i.e. P ⊆ NP. If a problem is solved in polynomial time by deterministic Turing machine then that problem is also solvable by non-deterministic Turing Machine in polynomial time [19]. Informally, we can say the NP is the set of decision problems which a "Lucky Algorithm" solved in polynomial time and makes a right choice from a set of known choices.

- *Definition No. 3 of NP*

A problem is in class NP if there exists non-deterministic polynomial algorithm to solve it. Or we can say that a problem L is in NP if and only if there exists a polynomial time verification algorithm for L.

Examples:
The Hamiltonian Cycle (HC) problem:

1. Input is a graph.
2. Does G have a Hamiltonian Cycle?

This problem can be solved by the following NP algorithm.

Hamiltonian_CycleP (G,V)

```
   1. Begin
   2.   /*  This  loop  is  for
guessing*/
   3.     for i ← 1 to n
   4.       do
   5.      X[i] : = select(i);
          /* select()is an imaginary,
                non-implementable
                instruction */

   6.        end_of_for


          /* verification stage */
   7.        for i   1 ← to n
   8.         for j  i ← 1 to n do
   9.        if X[i] = X[j] then
  10.            return (no);
  11.              end_if
  12.              end_for
  13.               end_for

  14.      for i   1 ← to n-1 do
  15. if(X[i],X[i+1])is not an edge
             then
  16.               return(no);
  17.                  end_if
  18.                  end_for loop
  19.      if (X[n],X[1]) is not an
edge
             then
  20.           return(no);
  21.             End_if

  22.           return(yes);
  23.               End
```

If we analyze the above algorithm, we see that the size is O(n), and the computation time of the verification stage is O(n^2). It shows that the solution time is polynomial and non-deterministic algorithm solves the HC problem. Therefore the HC problem is an NP problem.

Similarly an example of the NP problem is the K-clique problem.

1. The input is a graph G and an integer k.
2. Question: Does G has a k-clique?

A non-deterministic algorithm for the k-clique problem is as:

K_CLIQUEP (G,k)

```
 1. Begin
   /*  this  loop  represents
the guessing stage*/

 2. for i ← 1 to k
 3.    do
 4.         X[i] := select(i);
 5.           End_for

   /*  verification  stage  is
as follows */
 6.    for i ← 1 to k do
 7.    for  j ← i+1 to k do
 8.  if  (X[i]  =  X[j]  or
   (X[i],X[j])
      is not an edge
 9.      Then
 10.         return(no);
 11.           end_if
 12.       end_for
 13.        end_for
 14.         return(yes);
 15.     End
```

By analyzing the above algorithm, we get that the solution size of k-clique is O(k) i.e. O(n), and computation time for the verification stage is O(n^2) and the algorithm is non-deterministic. Therefore this is an NP problem.

In computation complexity theory there are problems which can be solved by polynomial time by programs or algorithms but these programs or algorithms don't run in polynomial time on a regular computer, but run in polynomial time on a nondeterministic Turing machine. The problems solved by these programs are in class in NP so nondeterministic Turing machine can perform everything as a regular computer can. It means that all problems in class P are also in class NP.

Is P = NP? It means that if a problem is solved in polynomial time can also be verified in polynomial time, and vice versa. It is $ 1, 000, 000 prize question in complexity theory offered by Clay Mathematical Institute [1]. If someone could prove this, it would be a revolution in the field of computer science because it will enable us to construct efficient algorithms for more complicated and important problems [1].

F. NP-Hard Problems

In computational complexity theory informally, NP-hard is a class of problems that are the hardest problems in NP. In many cases, we may solve a given problem by reducing it to another problem. More specifically, a Problem *P* is NP-hard when every problem *A* is NP can be reduced in polynomial problem to *P* [17]. As a result, if we find a polynomial algorithm to solve any NP-hard problem, it means that there are polynomial algorithms to solve all the NP problems, "which is unlikely as many of them are considered hard" [18]. This means if we can solve NP-hard problem, then any problem in NP can be solved easily. Consequently this would prove P = NP.

NP-hard problems frequently deal with rules-based languages in many areas including [20]:

- Approximate Computing
- Cryptography
- Data mining
- Decision support
- Planning
- Process monitoring and control
- Routing/vehicle routing
- Scheduling
- Selection
- Tutoring systems

In computation process, there are some problems for which the fast solutions are impossible. To solve these problems we translate a problem to another and then find fast solution. In NP-class for some problems, if every single problem is translated then automatically fast solution to such a problem would give a fast solution to every problem in NP. Such types of problems are in NP-hard class. Also there are some NP-hard problems that are actually not in NP.

G. NP-Completeness (NPC)

In complexity theory, if a problem is both NP and NP-hard then such group of decision problems is called NP-complete denoted by NPC-class. These are the hardest problems in NP set. Let B is a decision problem, then B is in class NPC if it satisfies the conditions below:

1) B is in NP (NP-complete problems can be verified quickly for any given solution, but no efficient well-defined algorithm exists for solution).
2) Every problem in NP set can be reduced to B in polynomial time.

Although the known solution of NP-complete problems can be verified in polynomial time but at the first place, there is no efficient method for NPC problems solution. Indeed, it is the most distinguished characteristic of NP-complete problems. It means that to solve the problems by using current algorithms, the computation time increases very quickly with the increase of problem sizes [20].

To prove that a problem may in NP-complete, first we prove that the problem is in NP, and then reduce some known NP-complete problem to it. There a variety of well known NP-complete problems. Some of these are shown in the following list.

- Boolean satisfiability (SAT)
- Travelling salesman (decision version)
- Sub-graph isomorphism
- Hamiltonian
- Subset sum
- Clique
- Independent set
- Dominating set problem
- Knapsack
- Vertex cover problem

H. optimization problems

Unlike decision problem that gives us one correct result for each input either yes or no, the optimization problem deals with best answer for a particular input. Naturally the optimization problem arises in many areas such as travelling salesman problem and linear programming [21]. Many standard tools are used to translate function and optimization problems into decision problems. Suppose, in the salesman problem, the optimization problem is to construct a tour with nominal weight. The related decision problem is: for each M, to decide whether the graph has any tour with weight less than M. By repeatedly answering the decision problem, it is possible to find the minimal weight of a tour [21].

Reduction

In computation theory, a reduction is the process of transforming one problem into other. The purpose of reduction algorithm is to prove that the second problem is at least as difficult as the first one. Let L_1 and L_2 are the two decision problems. From the figure 2, we see that x is an input for L1, inside algorithm for L1, the transform function f translate input x to f(x), input for the L2 algorithm and produce yes/no based on f(x). This is also the result for L1. The idea of reduction algorithm is to find a transformation from L_1 to L_2.

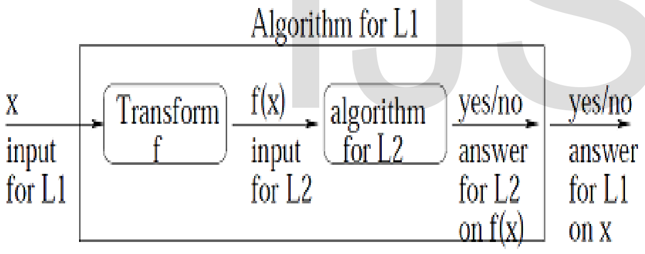
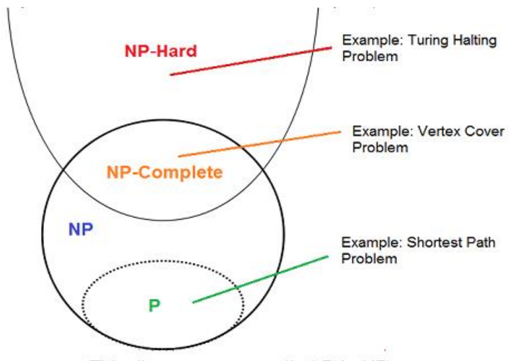


Figure: 02 Reductions L1 to L2

Reduction algorithm plays very important in computational theory. If we have already solved problem then by reducing a new problem to an existing one will save a lot of computation work. Suppose if a directed graph is given and we want to find minimum product path, where path is the multiplication of weights of edges along the path. If an algorithm for Dijkstra’s algorithm for shortest path already exists, then we take log of all weights existing Dijkstra’s algorithm to find the minimum product path there is no need to write new code.

Relation among the classes

The following figure shows the relationship of various computational complexity classes:


NP-Hard
Example: Turing Halting Problem
NP-Complete
Example: Vertex Cover Problem
NP
Example: Shortest Path Problem
P

This diagram assumes that P != NP

Conclusion and future work

This paper discussed various aspects of computation complexity classes. The purpose of this study is to find efficient algorithms to solve problems using tricky methods and techniques and avoid the process of the tedious and exhaustive search, with the help of the hints from the input in order to reduce and minimize the search space significantly [7]. For every NP-complete problem there exist some algorithm to solve but exponentially and on the cost of multiple resources hunting such as time and space. There are some problems which are still no computable or undecidable such as halting problems or Helbert’s Tenth problems. Mathematicians and researchers are continuously trying to explore fast and efficient algorithms and a problem will be unsolvable today but tomorrow may turn to be solved efficiently. Actually NP is a class of decision problems. Though one may informally talk about some problems being in NP, actually that doesn't make much sense, they are not decision problems. Some of these problems might even have the same sort of power as an NP-complete problem. An efficient solution to these (non-decision) problems would lead directly to an efficient solution to any NP problem.

References

1. Jessica Su, Stanford University, retrieved from www.quora.com. Dated 20/12/2016
2. Radoslaw Hofman, Poznan, “Why LP cannot solve large instances of NP-complete problems in polynomial time”, 2006.
3. Cook S.A., “The complexity of theorem-proving procedures”, Proceedings of the third annual ACM symposium on Theory of computing, 1971, pp. 151-158.

4. Diaby M., "On the Equality of Complexity Classes P and NP: Linear Programming Formulation of the Quadratic Assignment Problem.", Proceedings of the International Multi Conference of Engineers and Computer Scientists 2006, IMECS '06, June 20-22, 2006, Hong Kong, China, ISBN 988-98671-3-3.
5. Scott Fortin, "The Graph Isomorphism Problem, Technical Report",S TR 96-20 Department of Computer Science, The University of Alberta Edmonton, Alberta, Canada, July 1996.
6. Lance Fortnow, "The Status of the P Versus NP Problem," Communications of the ACM, Vol. 52 No. 9, Pages 78-86, September 2009.
7. www.cs.berkeley.edu/~vazirani/algorithms/chap8.pdf
8. Rene Peeters, "The maximum edge biclique problem is NP-complete", Department of Econometrics and Operations Research, Tilburg University, Tilburg, 5000 LE, The Netherlands, 2003.
9. D.S. Hochbaum, "Approximating clique and biclique problems", J. Algorithms 29 (1) (1998) 174–200.
10. D.S. Johnson, "The NP-completeness column: an ongoing guide", J. Algorithms 8 (3) (1987) 438–448.
11. M. Yannakakis, "Node- and edge-deletion NP-complete problems", Proceedings of the 10th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 253–264.
12. Radoslaw Hofman, "Why LP cannot solve large instances of NP-complete problems in polynomial time", Paznan2006.
13. Karmarkar, N., "A new polynomial-time algorithm for linear programming," Combinatorica 4, (1984) pp. 373-395.

14. Khachiyan, L.G., "Polynomial algorithm in linear programming,", Soviet Mathematics Doklady 20, (1979) pp. 191-194.
15. Yannakakis, M., "Expressing Combinatorial Optimization Problems by Linear Programs", Journal of Computer and System Sciences 43 (1991) pp. 441-466.
16. Computational complexity theory, Wikipedia, visited date: 20/01/2017
17. Leeuwen, Jan van, Handbook of Theoretical Computer Science. Vol. A, "Algorithms and complexity". Amsterdam: Elsevier. ISBN 0262720140, ed. (1998).
18. Daniel Pierre Bovet; Pierluigi Crescenzi. "Introduction to the Theory of Complexity". Prentice Hall. p. 69. ISBN 0-13-915380-2.
19. PHYS771 Lecture 6: "P, NP, and Friends". www.scottaaronson.com. Retrieved 10-01-2017.
20. "NP Hardness", Wikipedia, visited date: 20/01/2017
21. "Decision Problems", Wikipedia, visited date: 20/01/2017