Easy understanding of computational complexity theory – a student's perspective

Bhalchandra P.U.*, More U. L., Rawangaonkar R. R., Tammewar P. R., Kulkarni P.P.

*School of Computational Sciences, Swami Ramanand Teerth Marathwada University, Nanded, 431606, MS, India, pub1976@rediffmail.com

Abstract- Computer Science is largely concerned about a single question - how long does it take to execute a given algorithm? But computer scientists don't give the answer in minutes or milliseconds; they give it relative to the number of elements the algorithm has to manipulate resulting into formation of computational complexity theory. For entry level students in Data Structures and Analysis of Algorithms, understanding this theory is quite hard as literature review seen so far is dedicated to classifying problems by how hard they are. We have attempted for rigorous analysis of literature review and discovered different types of problems. A division of problem space is suggested in order to put these concepts in simple and conceptually understandable format.

Keywords- Complexity Theory, P, NP completeness

Introduction

Complexity analysis topic is familiar to every computer scientist, computer programmer, software engineer or related professionals. Every one of them has heard about it but can't provide a definition any better than an intuitive one, which is usually not very accurate. They say the same like what many people think. However P and NP are not the only two complexity classes, rather there are many of them; even hundreds of them are listed and explained on Stanford University's website under the link "Complexity Zoo". It's an extremely deep topic we could spend our whole life learning about. We feel that we don't need to know everything of it, but it's good to know the most common classes and their problem domains. We hope this work will be useful to clarify the doubts of entry level students on this topic.

Understanding Classification of Problems

After rigorous analysis of review literature, we have discovered different classifications of complexity problems. Here, the word complexity refers to the running time of an algorithm. These different classifications have one technical question in common. These are all really defined in terms of yes-or-no problems and often forcing researchers to introspect regarding does a certain structure exist rather than how does he find the structure?

a) P Problems - "P" stands for polynomial. These can be solved in polynomial time. These problems have formed the main theme of every advanced Data Structure and Analysis of Algorithm course.

b) NP Problems - "NP" stands for "Nondeterministic Polynomial time" where nondeterministic is just a fancy way of talking about guessing a solution. A problem is in NP if we can quickly (in polynomial time) test whether a solution is corrects and that too without worrying about how hard it might be to find the solution. Many students accept NP mistakenly.

NP does not stand for "Non-Polynomial". There are many complexity classes that are much harder than NP.

c) PSPACE Problems- This stands for Polynomial in SPACE .Problems that can be solved using a reasonable amount of memory - defined formally as a polynomial in the input size, without regard to how much time the solution takes.

d) EXPTIME- This stands for Exponential Time problems .This class contains most problems we are likely to run into, including everything in the previous three classes. We found that, this class is not all-inclusive and there are problems for which the best algorithms take even more than exponential time. A usual case is searching an item in a balanced tree where complexity becomes exponential as we penetrate deeper in the trees.

e) Undecidable. For some problems, we can prove that there is no algorithm that always solves them, no matter how much time or space is allowed. A usual case is finding a longest path between two nodes in a cycling graph.

We suggest a vertical division of problem space depicting these problems. As we move down, problem becomes harder and harder. This pictorial representation makes it ease for students to understand rather than reading reviewed literature rigorously.

Fig. 1- Diagrammatically dividing Problem space into types of Problems

Theoretical Perspective on Complexity Theory

Initially we mentioned that this all thinking is regarding Complexity Analysis of Algorithm. Algorithms are always judged by keeping in mind the "Solvability" and "Trying Efforts aspects". This complexity is always measured in Asymptotic Notations – called as Big-O (pronounced as "Order of-") notations. The Asymptotic measures are the only appropriate tools if we are dealing with big inputs. These measures allow us to compare two algorithms in exact analytical functions. With asymptotic complexity all we are interested in is the dominant term, in the limit of the efficiency function. It is evident from above discussion that Asymptotic are the foundations of computational complexity theory [2,3]. The solvable problem space is tractable in constant time. For an input state n , they result in Logarithmic Time $O(\log n)$, Linear Time $O(n)$, Quadratic Time $O(n^2)$, Cubic Time $O(n^3)$ and Higher Polynomial Time – Super Polynomial Time $O(n^4 \text{ to } n^k)$. These all categories constitute Polynomial Type complexity and commonly seen in searching, sorting, pattern matching problems. This is characteristics of P class problem. All the algorithms did so far have been polynomial time algorithms with worst case running time as $O(n^k)$. So these problems are regarded as tractable. The problems requiring time in higher degree polynomial may be regarded as intractable but there are very few that come in this category. A problem that can be solved polynomially in one model can also be solved polynomially in other model. The class of polynomial time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication and composition. E.g. if output of one polynomial time algorithm is put into another, the composite algorithm is polynomial. Although defined theoretically, many of above classes have practical implications. For instance P is a very good approximation to the class of problems which can be solved quickly in practice. For such problems; we can prove a polynomial worst case time bound indicating that corresponding algorithms are really practical. P is the class containing all the problems solvable, by a deterministic Turing machine, in a polynomial amount of time depending on the input size. This seems easy to understand to most of us. The NP, on the other hand, does not seem so straightforward to understand to everybody. The definition of NP is the same as given for P, but changing "deterministic Turing machine" for "non-deterministic Turing machine". Adding these 3 letters makes the machine more powerful, so NP includes many problems which are not in P (if $P \neq NP$). You can think of the non-determinism as if, every time the machine has to explore 2 different branches, it can explore both of them at the same time. You can also view it as if, when

having to explore 2 different branches, it always chooses a branch that leads to accepting (if such a branch exists). Another interesting feature of NP problems is that a solution can be checked in polynomial time by a deterministic Turing machine. So, checking a solution of a NP problem is a P problem.

Theoretical Perspective mainly NP Theory

We have often seen NP-Completeness theory as default part many courses on Data Structures and Analysis of Algorithms [1,2]. Many students fail to understand what does it mean? As they do not have undergone a course in empirical analysis of algorithms, they can not grasp the significance of lower, upper and worst bounds. This makes them unable to conclude with computationally feasible or not feasible type of problem. Now time has come to think about nonsolvable class, which is usually associated with NP measures. It has non deterministic view. If the result of every operation is uniquely defined then the algorithm is called deterministic algorithm. If we allow algorithms to contain operations whose outcome is not uniquely defined but is limited to a specific set of possibilities. This leads to the concept of non-deterministic algorithm. While dealing with problems, there could be unsolvable patterns. Here somebody has proved that we can not solve the problems. This category has the problem statement, problem input and solution states well defined. But how to get the solution is not defined. The Halting Problem is the most common example we found cited every where. A problem belongs to the class NP if the correctness of the problem's solution can be verified by a polynomial-time algorithm. Whether a given path of given length " k " exists between given " s " and " t " nodes in a graph " g " can be found using polynomial time. So actually we convert the problem into a decision problem and verification can be done in polynomial time. For NP-complete problems it may be possible to find the near optimal solution in polynomial time. An algorithm that returns near optimal solution is called as Approximation algorithm. Further, it is evident from Figure-1; a problem is NP-hard if it is as hard as any problem in NP. Here, the existence of a polynomial-time algorithm for an NP-hard problem implies the existence of polynomial solutions for every problem in NP. Similarly, NP-complete problems are the NP-hard problems that are also in NP. Here, the existence of a polynomial-time algorithm for an NP-complete problem implies the existence of polynomial solutions for every problem in NP. Till now no polynomial time algorithm has been discovered for any NP-hard or NP-complete problems. Many courses on Analysis of algorithms concentrate on NP-completeness.

When is a problem in the NP-complete class?

We propose two properties for easy judgment

1. It has to be a NP problem.
2. If it can be solved in polynomial time, then all of NP problems can.

The second is in itself a class: NP-hard. A problem can be NP-hard and not NP-complete, because it can be of more difficult complexity class than NP. In our view, NP-completeness theory is concerned with the distinction between the first two classes, P and NP. Problems in NP are still relatively easy and if only we could guess the right solution, we could then quickly test it. If a problem is NP, we can easily verify whether someone has found the right answer. We propose a simple analogy for easy understanding. If someone tells us that the combination to his mobile phone key pad security code is 1-2-3, then we can just dial up 1-2-3 and see if it opens? The important thing which must get cleared here is that, it's easy to verify whether we have got the right answer, but it's not necessarily easy or even feasible, to come up with the right answer in the first place. So, an example of an NP problem is, "find the combination to this mobile keypad." If students come up with a way to answer that question, it's easy for us to tell whether they are right or wrong. We must keep in mind that nobody has ever proved that some problems are fundamentally hard. So it's possible that there is a quick way to solve one of those NP-complete problems, and we just haven't found it yet. But then there is the fact that people have been thinking about it for 50 years now, and all they've done is find more and more NP-complete problems and not a single fast algorithm. So for practical purposes, NP-complete problems are the ones that we know are hard, even though we can't prove that they're hard. We are indulged in such hard and easy controversy because, for over last 50 years, artificial intelligence (AI) has not fulfilled its promise of devising truly intelligent machines for general use. As early as the 1950s and 1960s, scientists developed computational models of intelligence. They, then excitedly coded these models into the best computers of the day. At first the scientists were puzzled by the machines' inability to produce reasoned output. Soon they became frustrated when they realized that actually they had banged into an unforeseen brick wall. This wall had stopped them in their tracks and continues to do so today. AI also has stymied scientists and engineers in other fields such as operations research -the field concerned with determining efficient manufacturing and scheduling protocols, VLSI chip design and testing, and data base management. The brick wall exists because many combinatorial problems that are fundamentally important are NP-complete.

Proposing Solutions to NP Complete Theory

Consider an example taken from manufacturing Industry. A general purpose welding device is to be used on an assembly line. It will make numerous welds at predetermined positions on a particular kind of part. The positions will be welded in a specific sequence called a schedule. This schedule is programmed into the welder before a "run." Possibly as many as 1000 welds can be scheduled for each part that passes on the assembly line. Clearly the speed, and thus the cost, of producing a product depend on how fast the welder can finish all its welds on each part and return to its starting point. But the welder's speed depends on the distances traversed from one position to the next. What is needed is an optimal schedule for the positioning of the weld-head: that is, a schedule that minimizes the total distance traversed by the weld-head to finish each part. Let's consider how we might find such an optimal schedule. A simple algorithm can be proposed.

1. Tabulate all possible schedules along with the total distance traversed
2. Search for the schedule offering the least total distance.

This seems like a perfectly reasonable approach. But when we try to implement it we run into the thick brick wall. The number of schedules that we have to tabulate can easily be seen to be the factorial of the number of positions needing welds. Let "n!" denote this number where "n" is the number of positions needing welds. If we need even as few as "30 welds", n! will be "30!" which is greater than 10^{30} . On a computer that can tabulate schedules at the rate of one per nano seconds, we can tabulate our task involving 30 welds in no less than $10^{30} \times \text{Nano seconds}$. But this is near about 30 years. Thus NP-complete problems are the ones that we know are hard, even though we can't prove that they're hard. However we can still propose a solution to it. Before we search for a better algorithm, we must understand what is holding us back. The problem we have with tabulating possible solutions is that, if "n" is increased by 1, the size of the table, and therefore the amount of time needed to find an optimal schedule, more than doubles. This phenomenon is known as the exponential explosion of complexity. An algorithm suffering from this phenomenon has a complexity at least 2^n . What we need is an algorithm with a complexity n^2 or, better yet, "n". If we had a scheduling algorithm of complexity n^2 , we could solve our scheduling problem for 1000 welds in less than 1 second with a Pentium processor. With a complexity of n we could get a result in less than 1 second on an IBM XT. The execution time of these processors is typically between 500 nano seconds to 2 micro-seconds, because of the Reduced Instruction Set Architecture (RISC), pipelining mechanism used in them. How can we

find an algorithm of complexity n^2 or n ? We can achieve this complexity if there is a way to iteratively extend a partial solution in such a way that the extensions do not deviate from the optimal solution.

Possible solutions to NP Complete problems

The NP-complete problems really come up all the time. Knowing they're hard, we usually stop beating our head against a wall trying to solve them, and do something better. The possible solution can include,

a) Use a heuristic: If you can't quickly solve the problem with a good worst case time, maybe you can come up with a method for solving a reasonable fraction of the common cases.

b) Approximation: Solve the problem approximately instead of exactly. A lot of the time it is possible to come up with a provably fast algorithm, that doesn't solve the problem exactly but comes up with a solution you can prove is close to right.

c) Exponential Solution: Use an exponential time solution anyway. If you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution.

d) Good Abstraction: Choose a better abstraction. The NP-complete abstract problem you're trying to solve presumably comes from ignoring some of the seemingly unimportant details of a more complicated real world problem. Perhaps some of those details shouldn't have been ignored, and make the difference between what you can and can't solve. In recent days research scholars are mainly relying on Computer System Architectural changes to find out solutions to NP complete problems. The students reading this paper may quickly guess - whether parallel processors are the answer?. Perhaps we can achieve real-time solutions to NP-complete problems by dividing the computation required over many, many processors. Parallelization can help but cannot solve the problem altogether. Assume that there are not enough atoms in the universe and every processor requires at least one atom. Let there are fewer than 2200 atoms in the universe. (This is just a guess.) A weld scheduling algorithm of complexity $2n$ distributed evenly over 2200 processors (the maximum possible) operating at supercomputer speed would compute for more than one century if n is only 235. Clearly, another approach is needed. Another possibility is to make machines faster. Unfortunately, we seem to run into a barrier here, too. For several reasons, it is unlikely that computers, as we know them, will ever be able to perform even an elementary operation in less than 10⁻¹ seconds (This is just 1000 times faster than the time we assumed for the supercomputer). Seldom, greedy methods are used to solve NP completeness problems. In

rare cases they yield a solution. Consider above mentioned scheduling problem. These methods always chooses a next weld point based on some minimum weld value for extending a weld schedule. The algorithm will work as,

- 1) Start with a partial schedule of one weld position chosen arbitrarily
- 2) Repeatedly append to the partial schedule the weld position that is closest to the last position in the partial schedule and that has not yet been added to the partial schedule
- 3) When all positions are in the partial schedule, return the partial schedule as a full schedule.

The complexity of this algorithm is n^2 since the distance between all pairs of positions must be considered.

Very often NP- completeness problems are solved by Reduction. Above shown pessimism can be extended to all NP complete problems. Unfortunately most interesting real-world problems are NP complete.

How to prove NP-completeness in practice

There are many problems already known to be NP-complete, and listed in the bible of the subject, Computers and Intractability -A guide to the theory of NP-completeness, written by Michael R. Garey and David S. Johnson, W. H. Freeman, in 1979 [3,4,5]. If we suspect a problem we are looking at is NP-complete, the first step is to look for it in Garey and Johnson. The second step is to find as similar a problem as we can in Garey and Johnson, and prove a reduction showing that similar problem to be easier than the one we want to solve. If neither of these works, we could try to find an efficient algorithm. Stephen Cook proved in 1972 that a problem called Circuit-SAT is at least as hard to solve as any other NP problem. To solve Circuit-SAT is to read a schematic for a piece of electronics and say whether there's any combination of inputs that will make it output something. Ever since 1972, people have been proving the same thing about a whole bunch of other problems, using an interesting technique called polynomial transformation. Polynomial transformation bears some resemblance to the CRASH-TEST proof. The idea is to argue that if someone were to give us a fast algorithm that solves our problem, and then we can take that program and turn it into a program that solves Circuit-SAT. And if our program is just as hard to solve as Circuit-SAT, then, like Circuit-SAT, it must be just as hard as any other NP problem. So we've got two problems that are NP-complete, and we can prove a third and a fourth the same way. This statement is not at all digestible to students. We propose a simple analogy for it. There's a set of Dogs, and a set of Cats. Some animals are Dogs, and some are Cats. There's

no animal that is both a Dog and a Cat. There are some animals that are neither dogs nor cats. If an animal is in the set of Cats, then it is not in the set of Dogs, and vice versa. Any given mouse is not a Cat. The set of Cats itself, is not a Cat. And the set of Dogs is not itself a Dog.

Relationships between different classes

Fig. 2- Relation between P and NP

Some famous examples of Computational Complexity Problems

Example 1: Long simple paths.

This is a problem from Graph Theory .We don't know whether this problem is in P as we haven't told a good way for finding such a path .And in fact this problem is NP-complete, so we believe that no such algorithm exists.

Example 2: Cryptography.

It is formalized as an NP problem. Here we simply want to find a key for which code= RSA (key, text). If given a key, we can test it by doing the encryption ourselves, so this is in NP. The hard question is, how do we find the key? For the code to be strong we hope it isn't possible to do much better than a brute force search.

Example 3: Chess.

What is involved in chess programming? Chess playing programs work by traversing a tree of possible movements and there by finding what the possible consequences would be of each different move. We can traverse the tree of chess positions and just need extra memory to hold all moves. So perfect chess playing is a problem in PSPACE.

Example 4: Halting problem.

Suppose we have written a program, and start to run it. After five minutes, it is still going. Does this mean it's in an infinite loop, or is it just slow? It

would be convenient if our compiler could tell us that program has an infinite loop. However this is an Undecidable problem: there is no program that will always correctly detect infinite loops.

Conclusion

We have elaborated some concepts in the computational complexity theory. A possible division of problem space is also elaborated. Our approach is simple and lucid for easy understanding. NP-complete problems are also defined in a precise sense as the hardest problems in P. We also discovered that there are problems that are in NP, not known to be in P, and not likely to be NP-complete. The theory of NP-completeness is a solution to the practical problem of applying complexity theory to individual problems.

References

- [1] Michael Goodrich (2001) Algorithm Design: Foundations, Analysis, and Internet Examples. Wiley, ISBN-10: 0471383651.
- [2] Karam P.A. (2001) Stephen Cook Advances Knowledge of NP-Complete Problems, Assisting Computer Scientists: An entry from Gale's Science and Its Times, Gale.
- [3] Garey M.R. and Johnson D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, 1979, ISBN-10: 0716710455.
- [4] Michael T. Goodrich (2002) Algorithm Design, Foundations, Analysis & Internet Examples 2002 publication. John Wiley & Sons, ASIN: B00281EHY2.
- [5] Teofilo F. Gonzalez (2007) NP-Completeness: Theory and Applications. University of California, Santa Barbara.
- [6] Frank Hoffmann, David J. Hand, Niall Adams, Douglas Fisher, Gabriela Guimaraes (2001) Advances in Intelligent Data Analysis: 4th International Conference, IDA 2001, Cascais, Portugal, Springer, ISBN-10: 3540425810.

Table 1- Applications of NP- complete Theory- *The NP completeness theory is a magical stick for many types of problems [5,6]. We briefly summarize them in below table*

Packing Problems	SAT Problems	Internet
Routing Problems (TSP, Vehicle, etc)	Flow Problems	VLSI & Rectilinear Steiner Trees
Trees (Steiner, spanning, etc.)	Bioinformatics	Image Processing
Graph Problems	Superstring and Super sequence Problems	Logic -Computational Geometry
(Selecting , Partitioning , Arrangements , Embedding , etc)		Games and Puzzles
Protocol and Security	Data Bases, Query Languages, and Mining	Message Routing (computer, optical, wireless and sensor)