Sadhana, Vol. 19, Part 6, December 1994, pp 985-994. © Printed in India

Rudiments of complexity theory for scientists and engineers

V VINAY

Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India

Abstract. Complexity theory is an important and growing area in computer science that has caught the imagination of many researchers in mathematics, physics and biology. In order to reach out to a large section of scientists and engineers, the paper introduces elementary concepts in complexity theory in a informal manner, motivating the reader with many examples.

Keywords. Complexity theory; computational complexity; algorithms.

1. Introduction

All of us know how to calculate the *determinant* of a 4×4 matrix. We would use the standard row expansion with alternating signs. Using this algorithm involves computing the determinant of four 3×3 matrices or 12 2×2 matrices. The number of 2×2 matrices we need to compute becomes rather large as we increase the dimension of the matrix. In fact, an n-dimensional determinant would need $n!/2$ evaluations. However, we are also familiar with an easier way to solve the problem: use elementary row and column transformations to convert the given matrix into an upper triangular matrix and then simply multiply the diagonal entries. It can be argued that effort in terms of the number of multiplications and divisions involved is no more than n^3 operations. Though we know two ways to solve the same problem, we intuitively know that the latter method is preferable for large-dimensional matrices. Because the *complexity* in the number of steps is smaller.

Consider now a slightly different problem, the problem of computing a *permanent* of the same matrix. Its definition is similar to that of the determinant but with one important change: ignore the alternate sign change convention. Instead, all signs are taken to be positive. The row expansion method solves the problem but is again costly. Unfortunately, the elementary row and column transformations do not preserve the permanent of the matrix. We are now stuck with a very time-consuming method. Is there a better, more efficient way of calculating the permanent? Nobody

*Part of the work was done while the author was with the Centre for Artificial Intelligence and Robotics, Bangalore knows of any substantially more efficient method for doing so. Probably there are none! Can such statements be made with our current knowledge?

These are the nature of questions we want to address in this article. The article can be read by anybody who is exposed to elements of programming and who has used a computer. All concepts are introduced through examples, as far as possible. There are very few book on the subject. Readers who want to pursue their reading further can refer to Garey & Johnson (1979), Cook (1985) and Balcazar *et al* (1990).

2. Where does complexity theory stand?

The role of any scientific theory is to interpret and predict phenomena within its realm. Scientific theories act on a well-defined domain within which it interprets and predicts. A botanist finds a natural domain in flora and fauna. Mathematics creates abstract domains to investigate. Thus, for example, space need not be Euclidean. The domain may, of course, itself undergo refinements over time to accommodate for a deeper understanding. Thus, the domain of Newtonian mechanics is different from the domain of quantum mechanics, in that the latter refines the former in dramatic ways. In short, natural sciences seem to have "God given" domains whereas mathematics seems to create its own domain as and when required. A criteria often used is that natural sciences have observable domains. I wish to put forth the view that *computing is a natural one.* Unlike in mathematics, it is not created by axioms. And unlike in natural sciences, it does not have a physically observable domain. But the domain of all problems, is a "God given" one; and to that extent *the science of computing is a natural science.* And to the extent that the domain exists in the abstract, *it is mathematics.* Either way, the *science of computing* is a science, worthy of attention and contemplation by the best of minds.

Depending on one's perspective, *complexity theory* either subsumes or is subsumed by the science of computing. Complexity exists in every field. For example, just as the structural properties of the determinant was useful in designing an efficient method to evaluate the determinant, structural properties of chemicals can be used to speed the chemical process by using (for example) a catalyst. For the purposes of this paper, we shall restrict ourselves to the world of computing.

It should be emphasized that the complexity of solving a problem is *inherent* to the problem *per se* and does not depend on the nature of the computer we run the algorithm on, the programming language we use, and so on. The science of computing exists irrespective of the notion of a computer just as the notion of time is independent of a clock. Computers and clocks are realizations that give form to the abstractions involved.

3. Questions in complexity theory

One of the main spin-offs of any reasonable theory is that it provides a framework to classify or group like objects. Typically, the classification depends on a small set of parameters. Complexity theory is no different. First, a model of computation is fixed. Resources are identified in the model of computation. By varying the resources, different *complexity classes* result. Essentially, a complexity class (associated with a model of computation) is the set of all problems that can be solved on the model within the stated resource. Not all models of computation have physical realizations. However these models are important because they capture the essence of the complexity of the problem. We will quantify the sentence in a later section.

We will now list some of the main questions that require to be addressed. They are,

- How fast can a problem be solved?
- How fast can a problem be solved with the given resources?
- How do resources define complexity classes?
- Is the complexity class so defined robust?
- Is the problem intrinsically difficult to solve? If so, in what way?
- What properties does a complexity class satisfy?
- What can be said about relationships among complexity classes?

The first two questions are addressed by *design and analysis of algorithms*. The other questions come under *complexity theory*.

We have to explain some of the terms used above. When can we regard a complexity class as robust? As was explained above, a complexity class is the set of all problems that can be solved on an associated model of computation within some resource bounds. It is possible that a different model of computation with appropriate resource bounds on it also defines the same set of problems. In a happy situation such as this, we realize that the problems in the set that form a complexity class is "model independent", i.e., the intrinsic property (or properties) shared by the problems in the set is *not* specific to a model of computation. We can now say that a complexity class has alternate characterizations. Most complexity classes are robust. In fact, alternate characterizations of some complexity classes are positively startling and totally unexpected.

Alternate characterizations are also useful as they provide new insights into the problem and its complexity. These insights are valuable in identifying key structural properties of the problem in hand.

What do we mean by a problem being difficult to solve? In order to go any further, we need some definitions.

We imagine all our problems are encoded in binary. A problem instance is then a binary string. For technical reasons, we confine ourselves to the so-called *decision problems*: problem instances whose outcome is a "yes" or a "no". The "yes" instances of a problem constitute a set L. We will refer to L as either a problem or a language. In many complexity class, a notion of *completeness* can be introduced. A problem is said to be complete for a complexity class if

1. it belongs to the complexity class, and
2. it is as *hard* as any other problem in the class.

If $\mathcal{C}$ is a complexity class and L is said to be $\mathcal{C}$-complete if L satisfies the two properties listed above. If, however, only property (2) is satisfied, L is said to be $\mathcal{C}$- hard. The notion of being as hard as any other problem is technical – I will only briefly sketch what it means. Let L_1 and L_2 be two languages. We say L_1 is *reducible to* L_2 if there is a mapping wherein every string in L_1 is mapped to some string in L_2 and every string that is not in L_1 is mapped on to some string not in L_2. A problem is hard for a class if every problem (or language) in the class is reducible to the given problem. The mapping that witnesses the reducibility should be "simple" or "easy".

Suppose L_1 and L_2 are $\mathcal{C}$-complete. Then, by definition, L_1 and L_2 are the hardest problems in $\mathcal{C}$; and so L_1 is as hard as L_2 and vice versa. We may conclude that the two problems have the same "degree" of hardness.

All this is important as it allows us to pick *one* representative problem from a complexity class and investigating its properties gives us insight into the complexity class. This also allows one to refine the definitions of the complexity class, resulting in a deeper understanding.

Not all complexity classes seem to admit of a complete problem. Fortunately, many interesting complexity classes do. It make things that much easier to investigate the class.

4. Five examples

We will look at five examples. These problems have been chosen for simplicity. And each of them illustrate an interesting point.

1. Consider an $m \times n$ grid. Our goal is to count the number of paths from a point s (lower left corner) to a t (upper right corner). The only restriction is that as we move from s to t, we may do so by either moving right or moving up. Therefore all paths have length $m + n$.

We want to count the number of paths from s to t. Elementary counting tells us that there are ${}^{(m+n)}C_m$ paths, or $(m+n)!/(m!n!)$. Let us now provide a slight twist to the problem. Let us cut off some of the edges in the grid, as illustrated in figure 1. How do we now calculate the number of paths from s to t?

If we were to patiently enumerate all the paths from s to t, we could then count the number of paths we enumerated. But we need to be really patient to make this method give an answer. To see this, consider a square grid of size 60. Let us cut a small number of connections in the grid. By removing only a few edges, we ensure that the number of paths in the present grid is nearly equal to the number of paths on the complete grid. On the complete grid, we know that the number of paths is ${}^{60}C_{30}$ which is about 10^{17}. Even assuming that a new path can be enumerated every nanosecond, the task would take about 3 years! A similar task on a 100×100 grid would take almost forever.

The crux of the matter is that the number of paths grows *exponentially* in the dimension of the grid. (Assuming a square grid of size n, it is easy to see that ${}^{2n}C_n \geq 2^n$.) All this makes the method a very costly way of solving the problem. We require to find a much simpler way. Luckily, there is such a method. Consider any grid point. We can reach this grid point in exactly two ways: through its left neighbour or its neighbour below. (It is possible that both the neighbours do not exist, but at least one will.) Assume that we know the number of paths from s to each of the two neighbours is known. Clearly, the number of points to the given point is the sum of the number of paths to its two neighbours. If a neighbour does not exist, it simply means we treat it as a neighbour with count zero. The number of paths from s to itself is 1. The interested reader can work out the details and try

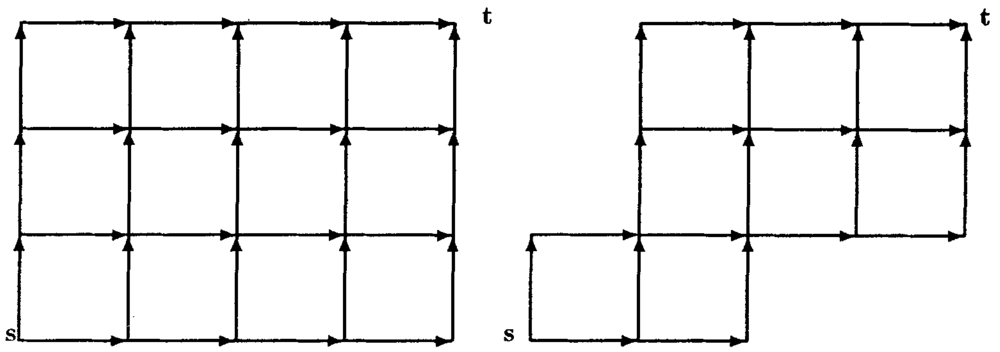


Figure 1. Counting paths between two nodes in a grid.

it out on a small example.

The number of additions to be carried out is no more than the number of grid points. This is because the moment we know the count of the neighbours of a point, we can determine the value of the point– so we need to look at each point exactly once. But a square grid of size n has n^2 points. If we now assume that each addition takes one nanosecond, we see that the number of paths on some 100-dimensional grid is no more than 10^{-5} seconds!

The reason for this improvement is because unlike the first method which takes *exponential time*, the second method takes *polynomial time*. Exponentials grow very fast and consequently many problems are almost impossible to solve, irrespective of the speed of the computer.

In an intuitive way, we can say a problem is *tractable* if it is solvable in polynomial time. That is, the graph plotting the size of the input to its time can be bounded above by a polynomial function.

2. Consider the problem of solving a jig-saw puzzle. All of us have solved one sometime or the other. In fact, it is common to find jig-saw puzzles with over a thousand pieces in the market. And they take hours to assemble. We want to find the complexity of solving this problem. All of us know the problem is not easy. The corners are easy to identify but in order to find pieces that go in the interior, we can do no better than trying out all possibilities until we find a piece that fits. This method is expensive and exhaustive. (You can easily convince yourself that there is an exponential effort involved.) Unfortunately, we do not know of any way to substantially speed up the time it takes to solve the puzzle.

One reason for this situation is that the problem appears to be lacking in structure that can help in finding a speedy solution. But it does have a property, a property that pleases and assures a complexity theorist– if somebody claims to have found a solution, you can easily verify the claim! I know this looks trivial, but hidden in it is one of the most profound ideas of our times.

The key is in the length of the solution as a function of the length of the input. In this case, the length of the input is the number of pieces. The length of the solution is linear (and hence polynomial) in the number of pieces. Let us write down two properties that the jig-saw puzzle has:

1. it has an easily describable solution;

2. the correctness of the solution is easily verifiable.

The word easy is used in the sense that the functions are polynomials. To us, polynomials are easy, tractable, efficient etc. In this example, we could not come up with an efficient method to solve the problem. But we identified two properties that the jig-saw puzzle satisfied. We replaced the criterion of "easily solvable" to "easily verifiable".

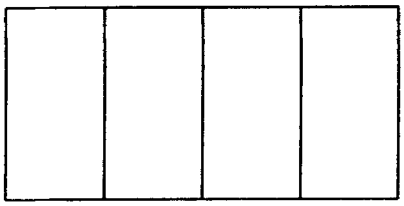
(3) Consider a solution to the previous example. Suppose it were written on a long sheet of paper and you are allowed to read it one word at a time (You can only remember a small number of words at any time). In the process of verifying the correctness of the solution, would you have to go back and forth over the written text? Let us see. Suppose there is a piece with three other pieces adjoining it. You verify that these are the three pieces that completely fit the fixed piece. Now you have to explore each of the three pieces further for correctness. While verifying the correctness of one of the three pieces, you would have to go back to the fixed piece as the fixed piece is a neighbour. You may argue that there is no need to check this up as it has already been verified. First of all, you cannot remember all the pairs you have checked as you have a small memory. (Before somebody jumps at me, I should state that I assume that each jig-saw has a constant number of neighbours; without this assumption, you may not even be in a position to verify the correctness of one fixed piece with respect to its neighbours!) Also, a moment's thought will convince you that this will allow your friend (who claimed the solution) to cheat! So it looks like we require to go back and forth on the presented solution. We will now consider an example where this need not be so.

The city of Bangalore has suddenly become a motorists' nightmare, especially after half of its major roads were made one-way. Some motorists are even scared of going to some localities because they may not be able to figure out a way to get out! (In the next example, we will show that their fears are not unfounded.) In such a case, it is advisable to carry a route they need to take to reach their destination, at least on their first visit. The instruction written on a piece of paper contains descriptions of the following form: "As you go down the road, take the third right" or "At a circle, take the fifth road in clockwise direction". It is important to note that the motorist need never go back to a previous instruction; (S)he has to only remember the current instruction. The solution in this case is rather appropriately said to be *one-way.*

Problems with one-way solutions are far more easier than problems with "back and forth" solutions. In particular, such problems can also be solved in polynomial time. Problems with "back and forth" solutions are not known to be tractable (i.e., have polynomial time algorithms). This is one of the biggest open problems in complexity theory.

(4) We will consider the effects of making roads one-way. You as a motorist move from one road to another in a random fashion in a city without one-way roads. By random, we mean the following: if you are at an intersection and say six roads meet there, you will take a road with probability 1/6. You will eventually reach your destination, of course. By an interesting application of probability theory, it can be proved that the number of steps (assuming one unit of time to go from one intersection to another) it takes on an average to reach the destination is a

polynomial in the number of intersections (or vertices in the graph). The proof is based on a simple analysis of a Markov chain.



We will now show that the number of steps need not be small in the presence of one-way roads. Consider a city with intersection and roads as represented by the graph above. Suppose we take a random walk starting at 0 and want to reach 9. A simple calculation shows that the probability of doing so is 1/8. In a graph with $n + 1$ vertices in each row, the probability works out to be $1/2^n$. So the expected number of steps required to reach the destination is exponential! This justifies the remark that was made in the previous section.

This example shows that randomness is also an useful computational resource. It can also be used to model the complexity of certain problems. The next problem shows another problem where randomization helps.

(5) Are the following two polynomials identical?

- $(3x^2 + 5x + 2)(4x^2 + 17x + 15)$
- $(x^2 + 4x + 3)(12x^2 + 23x + 10)$

If you expand each of the polynomials, you will require 18 multiplications and many additions to add like terms. In general, if the polynomial was a product of n arbitrary binomials, multiplying naively will result in 2^n terms. One can be smarter and regroup like terms at each intermediate stage. But it is still a lot of work. We would very much like to have a simpler scheme. Assume the degree of the polynomial is n. Fix a large number, say 2^n. Pick randomly a number between 0 and 2^n. Evaluate the two polynomials at this random number. If the answers match, declare the polynomials as identical. Otherwise declare them as different.

When the evaluated answers are different, the polynomials are obviously different. However, it is possible that two different polynomials give the same answer at the point we choose. But no two distinct polynomials can agree on more than n points. Here is where randomness comes in. The probability that we pick one of these n bad points is $n/2^n$-a really small probability! This example illustrates how randomization helps in simplifying algorithms. Randomness is indeed a useful resource.

5. A simple model of computation

A *Turing machine* is a standard model of computation. I will not give a formal definition of a Turing machine, but verbally describe it.

A Turing machine

- has a mechanism to access the given input. Through this mechanism, we can read any any bit of the input. The input cannot be tampered with.

- has a blackboard to do its calculations. The blackboard can hold a finite amount of information at any time. The size of the blackboard is the number of bits it can hold.
- a set of very simple rules (the program) which determine the input bit to be read, the part of the blackboard to be read, and the changes that need to be made.
- whistles[1] when the computation is complete and it accepts the input.
- whistles differently when the computation is complete and it rejects the input.

Time and space are two common resources defined on a Turing machine. By one unit of *time* we mean executing one step of the program. So time taken by a Turing machine is the total number of steps it takes before it whistles. By *space* we mean the maximum number of bits written on the blackboard. This determines the size of the blackboard we require for a particular problem. (Note that the size may vary with the length of the input.)

We would like to look at other models of computation. Some of these models are attractive as it is easier to reason with them. Specifically, boolean circuits provide a rich framework to model computation. A boolean circuit is composed of many gates– typically OR gates, AND gates and NOT gates. These gates are interconnected among themselves in some manner. There are also some gates called the INPUT gates which feed inputs to some gates in the circuit. The circuit has a distinguished gate called the output gate. The value of the output gate determines whether the input is accepted or rejected. By convention, we assume that NOT gates are at the input level, i.e, the input to a NOT gate is a circuit input. One constraint that the interconnections should follow is that there cannot be cycles or feedbacks. Basically, this means that computation on a circuit has a sense of direction.

Two common resources defined on circuits are size and depth. The *size* of a circuit is the number of gates in the circuit. The depth of a circuit is the longest path among all paths from the output gate to any input gate. Unless otherwise stated, we will assume that the gates of the circuit have unbounded fan-in. By a *semi-unbounded circuit* we mean a circuit where-in the OR gates have unbounded fan-in and the AND gates have restricted fan-in. The restriction on fan-in depends on the complexity class we want to characterize. The notion of semi-unboundedness introduces an asymmetry in the roles played by the gates in the circuit.

A single circuit has a fixed number of inputs, whereas problem instances have varying length. Therefore we require circuit families - one circuit for inputs of each length.

(1) The class $\mathcal{P}$ is the class of all tractable problems. Problems in this set are problems that can be solved in polynomial time on a Turing machine. Equivalently, they correspond to polynomial-sized circuit families. Important problems in $\mathcal{P}$ are Linear programming, Network flow, Horn clause satisfiability. All these problems are complete for $\mathcal{P}$.

(2) The class $\mathcal{DLOG}$ is the class of all problems that can be solved in $\log n$ space (n is the length of the input). This means that the size of the blackboared is restricted to be logarithmic in the input length. Since only a logarithmic number of bits can be

[1] A multimedia Turing machine!

written on the board, there are only a polynomial number of distinct messages that can be written on the board. This fact can be used to see that $\mathcal{DLOG}$ is a subset of $\mathcal{P}$. An important problem in this class is the evaluation of arithmetic expressions.

(3) The class $\mathcal{NP}$ is the class of all problems that have polynomial length proofs (or solutions) which can be verified by a $\mathcal{DLOG}$ machine. In fact, the verification machine can also be a $\mathcal{P}$ machine - this does not add to the power of the class. However, $\mathcal{DLOG}$ is preferable due to technical reasons. In terms of circuit, $\mathcal{NP}$ corresponds to constant depth exponential size semi-unbounded circus wherein the AND gates have polynomial fan-in. Note that the OR gates, by virtue of having unbounded fan-in can essentially have exponential fan-in. Important problems in $\mathcal{NP}$ are Integer programming, Travelling salesman problem, Scheduling, VLSI routing and Satisfiability in propositional logic. All of these problems are $\mathcal{NP}$-complete.

(4) The class $\mathcal{NLOG}$ is the class that corresponds to a "one-way" proof given to a $\mathcal{DLOG}$ machine as auxiliary input (along with the regular input). It is the set of all problems that can be solved by polynomial sized circuit families that is composed only of OR gatês. Important problems in this class are 2CNF satisfiability, Topological sort, source-destination connectivity.

(5) A very important parallel class is $\mathcal{NC}$. It is the set of all problems that can be be solved by a family of polynomial sized circuits with poly-log depth. By poly-log we mean $\log^{O(1)}(n)$. Circuit depth is easily seen to model hardware time. If we assume that the circuit is nicely layered, we see that a parallel machine can evaluate the circuit layer by layer. This definition of complexity is very idealized. For example, communication costs and delays etc. are not taken into consideration. But if a problem cannot be solved in even such an idealized model, there is no hope of solving the problem on a parallel computer. Note that $\mathcal{P}$ has circuits that could have polynomial depth. Important problems in $\mathcal{NC}$ are sorting, merging, shortest paths, recognition problems concerning chordal graphs, orthogonalization of basis vectors, determinant, rank of matrix, eigenvalue computation. In fact, $\mathcal{NLOG}$ is a subset of $\mathcal{NC}$.

(6) In $\mathcal{NP}$ instead of looking for a proof or solution to augment the input, we could also count the number of valid solutions. For example, the property of a graph having a Hamiltonian cycle is in $\mathcal{NP}$. The witness is a cycle in a graph which visits every vertex exactly once. Instead, we can count the number of hamiltonian cycles in the graph. In general the counting version of $\mathcal{NP}$ problems form a class called $\sharp\mathcal{P}$. A very important problem in this class is that of computing the permanent. This problem is in fact complete for $\sharp\mathcal{P}$. It has already been mentioned that the determinant is in $\mathcal{NC}$, which is within $\mathcal{P}$. The class $\sharp\mathcal{P}$ is now known to be a "big" class. So it is unlikely to have polynomial time algorithms. In this manner, complexity theory provides a new perspective to the determinant vs permanent question.

6. Conclusion

The main motivation for writing this paper is to create an awareness among scientists and engineers– that there is a very fundamental area of research in Computer Science, which is relevant to other branches of science and engineering.

The paper is informal in style and motivates the reader with many examples. The paper deals with the most elementary ideas in the field. Randomization is mentioned but no randomized complexity class is introduced in the paper as the definitions are technical. Other advanced topics are outside the scope of the current paper. It should be added that the area has been around for twenty-five years and the last five years have seen some breathtaking results in this area.

References

Balcázar J L, Díaz J, Gabarró J 1990 *Structural complexity* (New York: Springer-Verlag) vols 1 & 2

Cook S 1985 A taxonomy of problems with fast parallel algorithms *Inf. Comput.* 64: 2-22

Garey M R, Johnson D S 1979 *Computers and intractibility – A guide to the theory of NP-completeness* (San Francisco: Freeman)