Algorithms

9. Universality and Incomputability

R K Shyamasundar

In the previous articles, we had studied various topics such as algorithms, programs, programming languages, and algorithm design techniques. We shall now take a look at notions such as algorithmic universality, computability, incomputability and also discuss algorithmic limitations.

R K Shyamasundar is a Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

Introduction

With advances in technology, more sophisticated and complex computers with seemingly increased power are being developed. Several questions arise:

- Does one have to resort to experiments in order to learn about the capabilities of the sophisticated computers being developed?
- Does one have to learn 'all about' these sophisticated computers for understanding their capabilities?
- Are there some notions of *intrinsic capability* that capture what can and cannot be done with the existing computers and the ones to follow? Are there some primitive models which will enable the understanding of the nature of computation and its limitations?
- Are there problems which can 'never be solved' irrespective of the resources one has?

Answers to these questions can be seen in the pioneering work of logicians and mathematicians such as A Church, E Post, A M Turing and A A Markov. These pioneers

Previous articles of this series were:
1. Introduction to algorithms, January 1996.
2. The while–construct, March 1996.
3. Procedures and recursion, June 1996.
4. Turtle graphics, September 1996.
5. Data types and their representation in memory, December 1996.
6. Algorithms for sorting and searching, March 1997.
7. Data structures: lists, queues, stacks and arrays, June 1997.
8. Algorithm design techniques, August 1997.

introduced and studied various computational models and provided answers to the above questions in a convincing way.

In the following, we introduce a variant of the *Turing machine* (named after the pioneering English logician Alan M Turing, see article by P S Thiagarajan in *Resonance*, Vol.2, No.7, July 1997) model called the *counter machine* based on which we can understand the universality of algorithms and its limitations. We have chosen this model as it is essentially a subset of the programming language studied in the previous articles.

Universality of Algorithms

After introducing the basic *counter machine*, we discuss the Church-Post-Turing thesis which shows the universality of the model.

Counter Machines

The model can manipulate non-negative integers stored in variables. It has the following instructions:

	Primitive Instructions
$X:=0$	set X to zero
$X:=Y+1$	Increment Y by 1 and assign it to X
$X:=Y-1$	Decrement Y by 1 (if Y is 0 then $Y-1$ remains 0 and assign it to X)
	Control Instructions
if $X=0$ *then goto* L	if $X=0$ then control is transferred to instruction labelled L otherwise, control transfers to the next instruction
I1 I2	Execute I1 followed by I2 (**Sequencing**)

The variables are called *counters* as the operations possible on them essentially permit counting. A program in the above machine is a finite sequence of the above statements, some of which may be labelled. The algorithms we have seen in our earlier articles such as adding numbers, Euclid's GCD algorithm, etc. can be programmed in the above counter machine. The question is: What class of algorithmic problems can be solved in these machines? The answer is *quite surprising.* Counter machines are capable of solving any *effectively solvable algorithmic problem.* That is, any algorithm that has been written or will ever be written in any programming language executable on any computer can be encoded on the counter machine. This is the essence of the famous *Church-Post-Turing thesis* (CPT) named after the famous logicians/mathematicians Alonzo Church, Emil Post and Alan M Turing who discovered these characterizations independently (Also see *Box 1*). In other words, there is a counter program that would give exactly the same output for the same input that is given by the original algorithm. The counter program may take a long time before it gives the answer but it will surely give the answer provided the original algorithm gives the answer. In case the original program does not terminate on some inputs, the corresponding counter program also does not terminate (that means, if the original program uses unbounded resources so will the counter program).

Counter machines are capable of solving any effectively solvable algorithmic problem.

The interesting feature to be noted is that the above characterization is called a *thesis* rather than a *theorem.* The reason is due to the fact that the notion of 'effective computability' is informal and imprecise. The thesis relates the precise notion 'computable by a counter machine' (or the abstract formal equivalent models proposed by the above pioneers) to the imprecise notion 'computable' through the plethora of programming languages designed or to be designed. This is one of the most deep and far-reaching statements put forward by the pioneers which sounds more like a wild speculation than a reality. Justification of the thesis lies in the fact that the different wide-ranging models proposed for capturing the intuitive notion of 'effective computability' by Church (called *lambda calculus*), Turing (*Turing ma-*

Box 1. Robustness Computability

The Church-Post-Turing thesis implies that the most powerful supercomputer, with the most sophisticated array of programming languages, interpreters, compilers, etc is no more powerful than a simple PC with a very simple programming language. Given an unlimited amount of time and memory space, both can solve precisely the same algorithmic problems. The incomputable problems can be solved by neither.

As a result of the Church-Post-Turing thesis, computable problems become extremely *robust*. That is, the computation becomes *invariant* under changes in the model of the computer or the programming language. Hence, designers/architects of new models/languages must find reasons other than the raw solving power to justify their recommendations since problems solvable on one are also solvable on the other and in fact, all are equivalent to the primitive abstract models introduced by Church, Post or Turing.

D Harel in *Algorithmics: The spirit of computing*

chines, Post (*production systems*), Kleene (*recursive functions*), Markov (*Markov algorithms*) have all been proven to be equivalent.

From the above thesis, we can infer that for any given algorithm there is an equivalent counter program. Again one encounters the following question:

- Does one need a different computer for each algorithm?

One of the most interesting consequences of the CPT thesis has been the construction of a *universal program* which has the capability of behaving like any other program. The universal program, say $\mathcal{U}$, accepts the description of another program $\mathcal{P}$, and its input $\mathcal{I}$, and simulates or behaves exactly like it. That is, $\mathcal{U}$ yields exactly the same results as program $\mathcal{P}$ on $\mathcal{I}$; $\mathcal{U}$ terminates on $\mathcal{I}$ if $\mathcal{P}$ terminates with $\mathcal{I}$ and gives exactly the same results as given by $\mathcal{P}$; further, $\mathcal{U}$ does not terminate on $\mathcal{I}$ if $\mathcal{P}$ does not terminate on $\mathcal{I}$.

A simple PC or a general purpose computer is essentially a universal program U (in some language L_0). It takes a program P, (in some language L_2) and its input I and behaves exactly like it. In other words, it is insensitive to the choice of a machine or language. It is in this sense that the computer is called *Universal.* In fact, it is not difficult to construct a universal program. A schematic diagram shown in *Figure 1* depicts the universality. In the context of the counter machine this essentially implies that it takes two positive numbers, the first an encoding of the program and the second an encoding of the input. In short, a simple PC can do what a most sophisticated computer can do provided it is given enough time and memory.

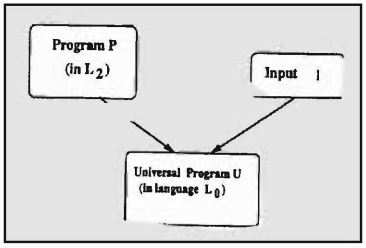

Figure 1. Universal program.

Computability and Incomputability

In the previous articles of this series, we have studied various algorithms. In this section, we show the existence of problems that cannot be solved algorithmically no matter what resources one has.

The following quotation[1] nicely elucidates the myth about the computer/software power:

> *"Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software"*

[1] This quote is extracted from a cover story on computer software in the *TIME* magazine of April 1984. (cf. D Harel, *Algorithms: The Spirit of Computing*).

A natural interpretation of the above statement is that the only reason for not being able to find a solution to a problem lies in the lack of resources such as time and memory. In this section, we show that there are problems that can *never be solved algorithmically* even with unlimited amount of resources. This would have deep philosophical implications in the sense that it not only shows the impossibility of arriving at a solution with unlimited resources but also

shows the limits of the artificial machines that have finite speed.

First, let us consider problems where we can convincingly say that algorithms exist:

1. Let us consider the simplest problem of finding whether there is a path from city A to city B in a given network of cities modelled as a graph. It must be clear that there must be an algorithm to answer this question since the number of paths from any node to another node in a finite graph is also finite (thus, one has to verify whether there is a path wherein the source is node A and destination is node B).

2. Consider the 8-queens' problem where one has to find all the configurations so that no queen can be killed by another queen. From the fact that the possible configurations are finite and each configuration can be checked for the property easily it follows that an algorithm exists for the problem.

3. Consider the problem of testing if a number is prime. Since any fixed number has only a finite number of divisors, we can easily say that there is an algorithm. It may be observed that unlike the earlier cases, the domain is not really finite as any number can be given as input and the same algorithm should work on all numbers.

In all the above problems, we can see that the configurations to be verified are bounded (in relation to the size of the input rather than being fixed a priori). Now, let us consider problems referred to as *tiling or domino's* problem. The tiling problem can be described as follows: We are given a finite number of 1×1 squares divided into four sections by the two diagonals, each quarter coloured with some colour (it is assumed that cards have a fixed orientation and cannot be rotated). An unlimited supply of these types of cards is assumed to be available. The problem is to find whether *any finite area of any size* can be covered by using the cards of

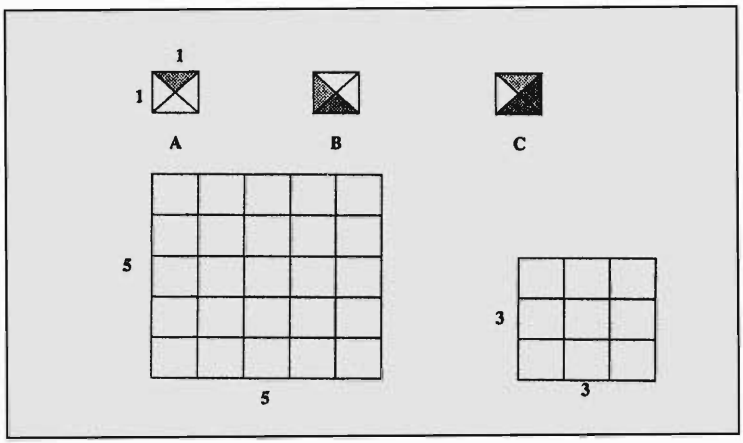

Figure 2. Tile types.

the given types, say as in *Figure 2*, such that the colours on any two touching edges are the same.

The constraint 'any finite area of any size' in the problem corresponds to the following interpretation: *All possible finite areas.* We are only excluding areas that are infinite. We can analyze the problem in the following way:

- With any given finite types of tiles, clearly we can answer whether a given area can be tiled since there are only finitely many configurations (it could be large but it is finite). The answer could be 'yes' or 'no'.
- If the answer was 'no', we can strengthen our finding to say: *With the given types of tiles, it is not possible to tile any finite area of any size.* The reason is that if we cannot surely tile some finite area, there is no way we can tile all possible finite areas.
- However, if the answer to the problem was 'yes', there is no way we can generalize our limited finding unless we exhaust all the possible finite areas (which are infinite) or come to some area which cannot be tiled.

The above argument can be summarized algorithmically as in *Figure 3.*

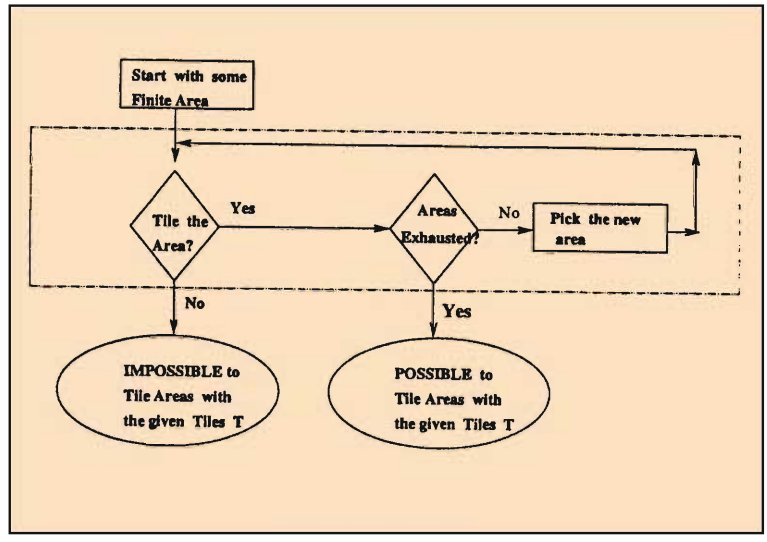

Figure 3. Flowchart for the tiling problem.

As the set of all finite areas is infinite, we can see that unless we find some area that cannot be tiled with finite set of types, the algorithm will go on forever. One possibility of having a solution is to treat the part of the flow chart enclosed in the dotted box as a 'basic step' itself; this would correspond to checking *if all finite areas can be tiled by the given types T.*

However, following our earlier discussion of the *basic steps* of the algorithm, the dotted box cannot be realized in any programming language as a basic step.

Any problem that does not have an algorithmic solution is called *incomputable* (or unsolvable). Thus, the above tiling problem is incomputable. If the problem calls for just *yes* or *no* answers only, the problem is called a decision problem. If the decision problem is incomputable, it is also termed *undecidable*; under this terminology, the tiling problem is undecidable. Another famous problem that has been proven to be undecidable is Hilbert's tenth problem[2] (see *Box 2* for history and details).

From the above discussion, there is a possibility of getting

[2] Diophantine equations with one unknown, i.e., equations of the form

$$a_n X^n + a_{n-1} X^{n-1} + \ldots + a_1 X + a_0 = 0$$

can be algorithmically solved. This follows from the fact that any integer solution X_0 must be a divisor of a_0. Since a_0 is a constant and bounded a priori, the algorithm follows immediately.

Box 2. Unsolvability of Hilbert's Tenth Problem

The renowned mathematician David Hilbert presented at the International Mathematical Congress in Paris in 1901, twenty three unsolved problems and directed the attention of mathematicians to these problems. One of the problems often referred to as Hilbert's tenth problem remained unsolved till Yuri Matiyasevič showed the problem to be *Unsolvable* in 1970. Hilbert's tenth problem is to give a computing algorithm to find out whether or not a given Diophantine equation (which is an equation of the form $P = 0$ where P is a polynomial with integer coefficients) has a solution in integers. Yuri Matiyasavič proved *that there is no such algorithm* in January 1970 using a beautiful and ingenious proof. He proved that no algorithm exists for testing a polynomial with integer coefficients to determine whether or not it has *positive integer* solutions. However, it follows from a well-known theorem of Lagrange that *every non-negative integer is the sum of four squares.* Thus, one could test the polynomial equation $P(X_1, \cdots, X_n) = 0$ for positive solutions $< X_1, \cdots, X_n >$ by testing $P(1+p_1^2, q_1^2+r_1^2+s_1^2, \cdots, 1+p_n^2, q_n^2+r_n^2+s_nr) = 0$ for integer solutions $< p_1, q_1, r_1, s_1, \cdots, p_n, q_n, r_n, s_n >$.

Unsolvability of this problem had been earlier reduced, through the works of several people such as Martin Davis, Julia Robinson etc., to the problem of finding a solution to an exponential Diophantine equation. An exponential diophantine equation is one in which powers of variables can themselves be variables, e.g. in addition to X^7, it also contains X^Z. Matiyasavič showed how the Fibonacci numbers could be used to construct such an equation.

Even though the result to the original question of Hilbert is negative, it has been resolved very much in the spirit of Hilbert's address in which he spoke of the conviction among Mathematicians *"that every definite mathematical problem must necessarily be susceptible for a precise settlement either in the form of an actual answer to the question asked or by the proof of the impossibility of its solution".*

Unsolvability of the problem leads to a strengthened form of Gödel's famous incompleteness theorem (see *Box 3*): *Corresponding to any given axiomatization of number theory, there is a Diophantine equation which has no positive integer solutions, but such that this fact cannot be proved within the given axiomatizations.*

an impression that the argument of undecidability of the tiling problem lies in showing the need of enumerating all finite areas in two-dimensions which is infinite anyway. From this one may conclude that it is essentially the *unboundedness of cases* that leads to incomputability. This is quite wrong. For example, let us consider the *Map Colouring Problem*:

> Let us say a *map* is any diagram in a plane divided into finite closed regions. Each finite closed region could be treated as a state or a country. The problem is to colour the map such that no two adjacent states share a boundary having the same colour. The question is: How many distinct colours are needed to colour any map?

It can be easily seen that there are infinitely many maps and if we use the argument of unboundedness of plausible

Box 3. Gödel's Incompleteness Theorems

Consider the statement: *This sentence is false.*
Is the above statement *true* or is it *false*?

Assume that it is true: in which case what it asserts must be true, which means that it must be false. But this contradicts our initial assumption. As a renewed attempt, assume that the statement is false: In which case what it asserts must be false, which means that it must be true. Once again, we have derived a contradiction to our starting assumption. We clearly have a paradox before us. We can assert neither the truth nor the falsity of such a statement with conviction.

The above statement is a variation of the popular paradox attributed to a certain Cretan named Epimenides, who said *All Cretans are liars.* The presence of such paradoxes in a natural language like English does not disturb us at all. In fact, such a possibility helps in expressing charming thoughts such as:

- *I used to think I was indecisive, but now I am not sure.*
- *Nobody goes to that restaurant any more, it is too crowded.*

Around the beginning of this century, it was noticed that such paradoxes also appeared in languages which were being used to formulate mathematics. One such

paradox in Georg Cantor's *Set Theory*, is attributed to Bertrand Russell. Consider the set N which contains all those sets which do not contain themselves. Does N contain itself ? A little thought shows that N contains itself if and only if N does not contain itself. This was a highly disturbing fact, and some of the most eminent logicians of the day set out to purge such paradoxes from mathematics. One of the foremost mathematicians of all time, David Hilbert, posed a challenge in 1900 to the mathematical community of the entire world. This challenge came to be known as *Hilbert's program*:

Reformulate mathematics in a precise system which satisfies two major criteria:

- *It should be possible to potentially prove all mathematical truths within the system* (completeness);
- *The system should be free from all paradoxes, and it should be possible to prove this fact in an incontestable manner within the system itself* (consistency).

In 1931 the celebrated logician Kurt Gödel announced his famous theorems which demonstrated that Hilbert's program was untenable. An inkling of Gödel's arguments may be gained from the following. Consider a formal system, let us call it X, which claims to satisfy the requirements of Hilbert's program. Gödel showed that within any such formal system X, it is possible to formulate a statement which proclaims:

This sentence is not provable in system X.

Let us try to determine the truth or falsity of the above statement. We immediately notice that it looks suspiciously similar to the Epimenides paradox discussed earlier. On the contrary, the above sentence does not lead to a paradox. Consider the case wherein the above statement can be proved within system X. Clearly, we have proved a falsehood. Since X satisfies the requirement of consistency, viz. X cannot prove falsehoods, this case is ruled out. On the other hand, if it is not possible to prove the above statement with system X, then clearly we have a truth and it is unprovable !

In effect Gödel demonstrated that any formal axiomatic system which satisfies certain minimum requirements of expressiveness, and is guaranteed to be consistent, is doomed to be incomplete. The system is incomplete in the sense that it fails to include all mathematical truths. Further, Gödel also showed that the consistency of a such system cannot be proved within the system itself, without lapsing into inconsistency !

cases, we would conclude it to be *incomputable*. In fact, for the above problem, it was thought that five colours were sufficient. It has been shown recently that the *Four Colour Conjecture* is indeed a theorem – thus establishing that four colours are sufficient for colouring all the possible maps.

The above example illustrates that undecidability cannot just be equated with unboundedness. Incomputability rests on the *problem of algorithmic termination* which is usually captured in terms of a problem referred to as the *halting problem*. Undecidability of other problems are established by reducing known undecidable problems to them. In the next article we shall discuss correctness of programs and also arrive at the proof of the undecidability of the halting problem.

Suggested Reading

- **B A Trakhtenbrot. *Algorithms and Automatic Computing Machines*. D C Heath and Company. Boston, 1963. One of the early books that nicely elucidates the notion of algorithms, effective computability and Turing machine models.**
- **C A R Hoare and D C S Allison. *Incomputability*. ACM Computing Surveys. Vol.4. No.3. pp. 169 178, September 1972. The article nicely elucidates the notion of incomputability through Russell's paradox using English adjectives.**
- **Douglas Hofstadter. *Gödel, Escher,Bach*. Penguin, 1979. Provides a general idea of Gödel's incompleteness theorems.**
- **D Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley Publishing Co.Inc, 1987.**
- **Raymond Smullyan. *Gödel's Incompleteness Theorems*. Oxford University Press, 1992. Provides detailed aspects of the incompleteness theorems.**

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax:022-215 2110